# Development of an API for creating and editing openEHR archetypes

Filip Klasson

Patrik Väyrynen

2009-02-12

LiTH-IMT/MI30-A-EX--09/472--SE

# Development of an API for creating and editing openEHR archetypes

**Master Thesis**

Institutionen för medicinsk teknik (IMT)
Linköpings universitet

**Filip Klasson**
**Patrik Väyrynen**

2009-02-12
LiTH-IMT/MI30-A-EX--09/472--SE

Supervisor: **Erik Sundvall** – IMT, Linköpings universitet
Examiner: **Daniel Karlsson** – IMT, Linköpings universitet

# Abstract

Archetypes are used to standardize a way of creating, presenting and distributing health care data. In this master thesis project the open specifications of openEHR was followed.

The objective of this master thesis project has been to develop a Java based API for creating and editing openEHR archetypes. The API is a programming toolbox that can be used when developing archetype editors. Another purpose has been to implement validation functionality for archetypes. An important aspect is that the functionality of the API is well documented, this is important to ease the understanding of the system for future developers.

The result was a Java based API that is a platform for future archetype editors. The API-kernel has optional immutability so developed archetypes can be locked for modification by making them immutable. The API is compatible with the openEHR specifications 1.0.1, it can load and save archetypes in ADL (Archetype Definition Language) format. There is also a validation feature that verifies that the archetype follows the right structure with respect to predefined reference models. This master thesis report also presents a basic GUI proposal.

# Table of contents

# Table of figures

## Declaration of relevant terms

| | |
|---|---|
| ADL | Archetype Definition Language is a Domain Specific Language (DSL) which textually expresses archetypes. |
| AM | The openEHR Archetype Model defines the structure and semantics of archetypes and templates. (2 p. 9) |
| AOM | Archetype Object Model, which expresses archetypes as objects according to the Unified Modeling Language (UML) (2 p. 9). |
| API | Application Programming Interface is a set of functions that helps when creating an application. All kinds of APIs exist for this purpose. I.e. sets of functions to facilitate communication with the keyboard, soundcard, network, manipulation of tree structures etc. |
| Archetype | Formal model of a clinical information entity. The model includes terms, restrictions and structure for patient data. The archetype is based on a Reference Model (RM). |
| EHR | Electronic Health Record is an individual patient's medical record in digital format. |
| Field | A field is either a class variable or instance member variable in Java. |
| IM | Information Model, which is a representation of concepts, relationships, constraints, rules and operations to specify data semantics for a certain domain (3). |
| ISO | International Organization for Standardization and it is the largest developer and publisher of international standards (4). |
| Method | What an object in Java can do is called methods (5 p. 34). |

| | |
|---|---|
| Node | In this project a node refers to an object in the definition part of the archetype. |
| Ontology | A shared vocabulary with objects and concepts that exist within a certain domain. |
| RM | The openEHR Reference Model defines the structure and semantics of information in terms of information models (IMs). (2 p. 9) |
| Semantic | In computer science semantics reflects the meaning of programs, data structures or functions. Programs can be described as having a syntactical part (grammatical and lexical structure) and a semantic part (meaning) (6). |
| Template | A template is when archetypes are merged to form a more complex structure like an examination of blood, where all the essential archetypes concerning this matter are included to build a new structure. |
| Terminology system | Terminology is the study of terms and their use. A terminology system is a set of terms used in a domain and the relationships between the terms. |
| XML | eXtensible Markup Language and its primary purpose is to help information systems share structured data. |

# 1 Introduction

## 1.1 Purpose

The purpose of the master thesis project was to develop an API for creating and editing archetypes. This API is intended to be used e.g. when creating graphical user interfaces (GUIs). The API is supposed to be a platform for future archetype applications like an archetype editor. One design goal was to make the code easy to understand by following specific code standards and writing Java documentation.

The primary goals of the project were to:

- Develop and implement an API for creating and editing archetypes.
- The API should include some validation functionality.

The secondary goals were to:

- Implement template functionality.
- Implement a simple GUI to better be able to show how the API works.

The secondary goals were only to be done if time permit.

There were only a few strict requirements given to this project and they were:

- The API should be implemented in Java.
- The API should be compatible with the openEHR specifications 1.0.1.
- The API should have mutable archetype objects.
- The code should be licensed as open source.

## 1.2 Methods and sources

When developing the API some of the thoughts from Extreme Programming have been used. The methods of the API have been developed according to an iterative process where testing has been an important part in successively improving the functionality of the API. (7)

Subversion (8) was used to have control and overview of the changes being made during the development phase. To have effective version control from within the Eclipse IDE the plug-in Subclipse (9) was used. TortoiseSVN (10) was used as windows client for files not related to programming.

A significant amount of time has been used to understand the openEHR specifications since the API should follow these structures. To get inspiration of how the API could be created some of the existing software for creating and editing archetypes, based on the openEHR specification, have been studied.

The main sources of information have been the openEHR specifications, where the Archetype Object Model (AOM) is one of the most important documents (2). Another important source was the openEHR Java reference implementation project lead by Rong Chen. The source codes of archetype editors have also been helpful. The most important of these editors has been LinkEHR. Other editors like the LiU-Editor and the Archetype Editor created by Ocean Informatics have also been an inspiration as well as source code from Zilics.

During the project there have been communications through e-mail with persons involved in the openEHR community. These e-mails have been used as sources in some areas during this project. Jose Alberto Maldonado and Diego Boscá Tomás working with LinkEHR have been helpful to give information about LinkEHR. Thomas Beale and Rong Chen have also been helpful through e-mail. A valuable source has also been the different mailing lists related to the openEHR community.

## 1.3 Typographical conventions

The specification documentation uses underscore "_" between each word in class and variable names while this report uses uppercase letters for each new word according to Java standard (11). For instance the class CComplexObject, the specification documentation would write C_COMPLEX_OBJECT.

This report writes Java fields with italics, for instance the field parent is described as *parent*.

## 1.4 Disposition

**Chapter two** gives a background to the area and explains the concepts that are used in the following chapters. The chapter gives an introduction to archetypes and how they are used. This chapter also describes the archetype object model and gives a background to validation of archetypes.

**Chapter three** describes the method that was used to develop the API and also how testing was done.

In **chapter four** the result of the project is presented. First the functionality of the developed API is presented followed by the validation process. The chapter ends with a description of the testing aspect of the project.

The **last chapter** is the discussion chapter where different aspects of the project development and planning are discussed, what design decisions were made and how they worked and also possible future development ideas.

# 2 Background

## 2.1 Archetypes

This section describes what an archetype is and also presents other information related to archetypes.

### 2.1.1 What is an archetype?

Archetypes are constraint-based models of domain entities and each archetype describes allowed configurations of data instances and the data instances are defined in a reference (information) model (2 pp. 7-8). The reference model contains valid instances of particular domain concepts. This project uses openEHR's information model as its reference model hence the constructed archetypes are called openEHR archetypes. All archetypes are expressed in the same formalism and also defined to be widely re-useable though they can be specialized to include local particularities (12 p. 8).

In medicine an archetype could be designed to constrain instances of a simple node/arc information model for example microbiology test result or liver function test (12 p. 9). Designed openEHR archetypes, expressed in ADL, can be found in the list of archetypes at openEHR (13). When mentioning archetypes in the context of electronic health records (EHRs) they can be simply explained as the structural and semantic specification of the elements that constructs the EHRs. For a more detailed definition about the archetype concept see Thomas Beale's archetype object model document (14).

### 2.1.2 Purpose of archetypes

Archetypes are created for many purposes that can be summarized with the following bullet list (12 p. 9):

- They allow domain experts such as clinicians to create the definitions which will define the data in their information systems.
- They provide runtime validation of data input.
- They provide a platform for effective and intelligent querying of data
- They give knowledge-level interoperability so systems can communicate with each other at the level of knowledge concepts

### 2.1.3 OpenEHR

The openEHR foundation is an international non-profit foundation that according to them self aims to enable an interoperable life-long electronic health record. Another goal of the foundation is to improve health care in the information society. (1)

To achieve these goals openEHR works with (1):

- Developing open specifications, open-source software and knowledge resources
- Engaging in clinical implementation projects
- Participation in international standards development
- Supporting health informatics education

### 2.1.4 Archetype Object Model (AOM)

The archetype object model is together with the archetype definition language (ADL) and the openEHR archetype profile (oAP) called the archetype model (AM). The AM is the base when it comes to defining the structure and semantics of archetypes and templates.

The openEHR AOM is defined in the package am.archetype, which is illustrated in

Figure 1: The openehr.am package with the AOM (archetype) package highlighted

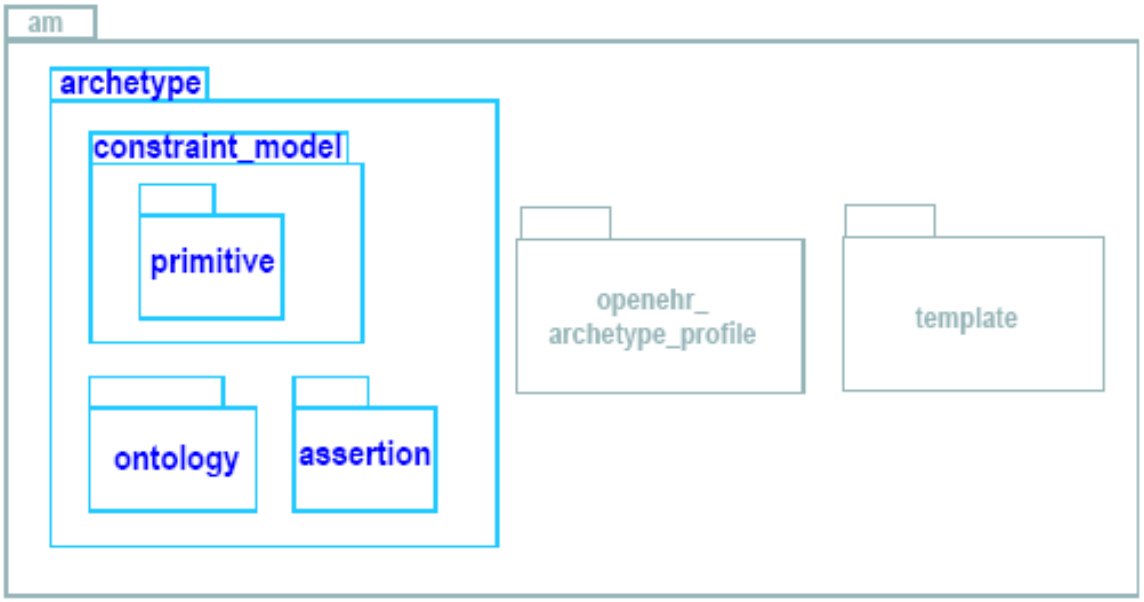


**Figure 1: The openehr.am package with the AOM (archetype) package highlighted (2 p. 9)**

An archetype object structure can be created in different ways. One way is to use a parsing process part of a programming language implementation that turns a syntax expression of an archetype (ADL or XML) into an object expression (2 p. 10). Another way is to use a programming language implementation of the openEHR specification that creates the objects that are defined in the AOM (2). This is either done by creating all objects at once which is necessary in an immutable model or by incrementally building the objects as is done in a mutable model. Information about mutability and immutability can be found in chapter 2.11. In this project that is Java based, the openEHR Java reference implementation project was used. When creating archetype objects from a syntax expression of an archetype the system converts an input file into an object parse tree with the help of a parser, the trees consists of the objects specified in the AOM. The process is illustrated in Figure 2.

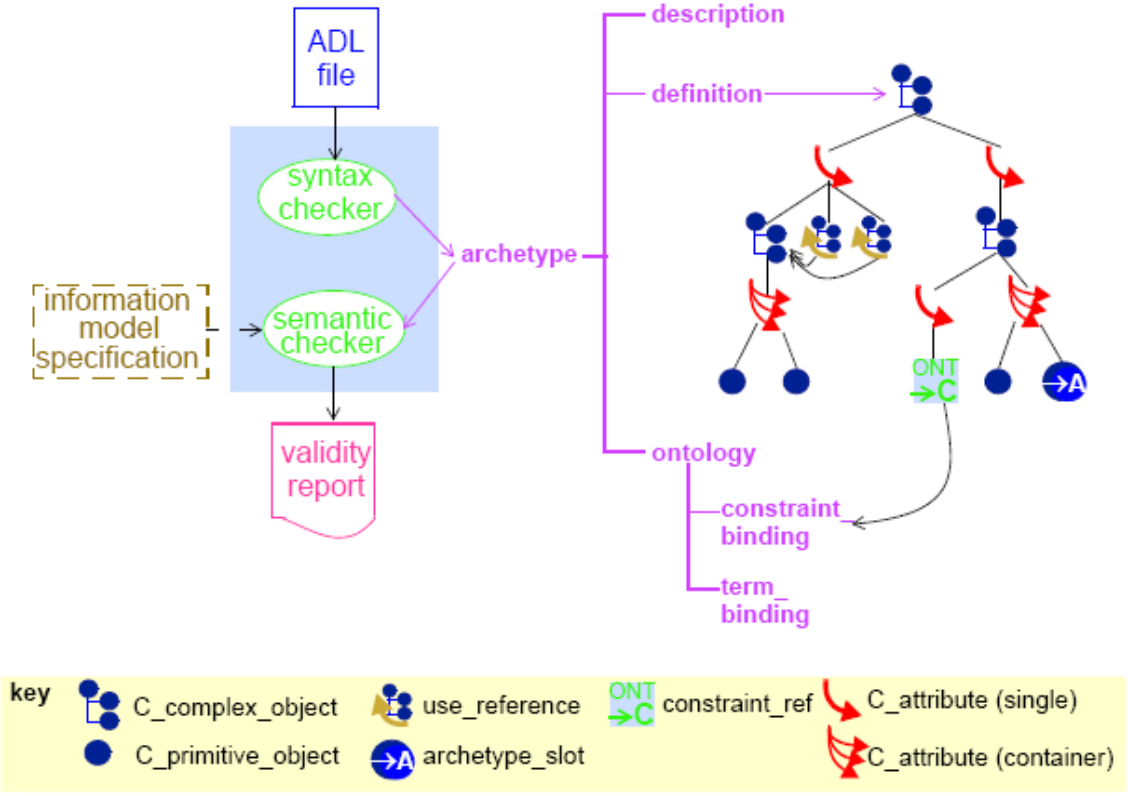


**Figure 2: The archetype parsing process to the left and the archetype structure to the right with the definition tree (1 p. 10)**

The archetype class extends the AuthoredResource class that includes descriptive meta-data, language information and revision history. In the archetype class one can find identifying information and it also includes definition, invariants and ontology for the archetype. There is also a utility

class called ValidityKind that functions with attributes whose value is expressing if the archetype is mandatory, optional or disallowed.

The archetype definition part contains an object tree whose root node is of the class CComplexObject. The tree has alternating layers of object- and attribute constrainer nodes, each containing the next level of nodes, see Figure 2. The leaves of the tree are primitive object constrainer nodes that constrain primitive types such as String, Integer, etc. Other objects that can be found at the leaves are ArchetypeSlot, ConstraintRef, ArchetypeInternalRef and children of CDomainType (COrdinal, CCodedText and CQuantity). These objects are described at the end of this section.

The invariants of an archetype are specified in an assertion class. These invariants mainly classify existence and validity of parts of the archetype, for instance if the description exists.

The ontology part of the archetype consists of an ArchetypeOntology object and it includes constraint- and term definitions. There are also constraint- and terminology bindings for connecting terms to external terminology systems. The definitions and bindings can be specified in many languages. The archetype package structure is described in Figure 3.

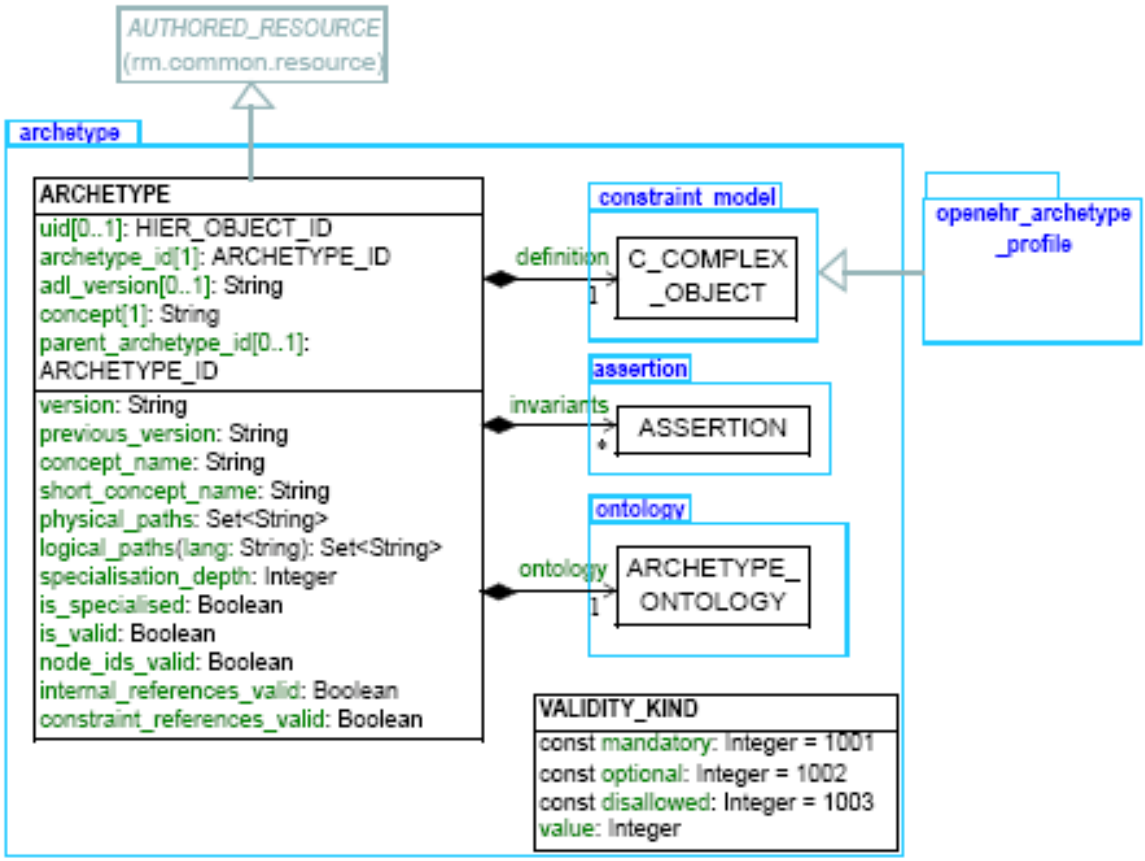


**Figure 3: openehr.am.archetype package (2 p. 13)**

### 2.1.4.1 Constraint Model

The openEHR constraint model defines the semantics of constraints of classes that are described in UML. A CComplexObject in the archetype definition part expresses constraints on objects described in the reference model. Each level of nodes in the tree that spans the archetype definition is narrowing the parent level of nodes. The root node is a reference model class that gets more restricted for each level of nodes in the tree and the leafs end up in restrictions on values like String, Integers, etc. The constraint model is illustrated in Figure 4.

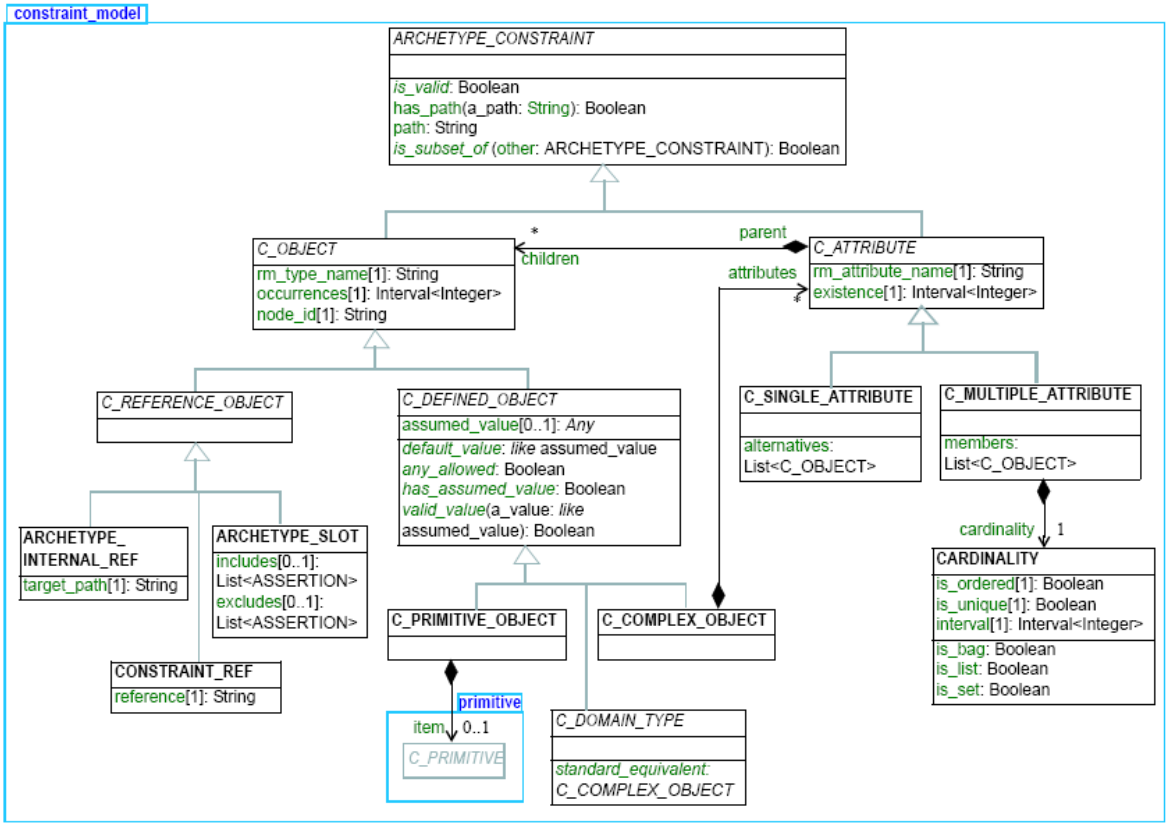


**Figure 4: openehr.am.archetype.constraint_model package that defines the constraint objects of an archetype (2 p. 18)**

The classes with bold font in Figure 4 are concrete objects and the other classes are abstract objects. The attributes of each class are also shown. The inheritance structure is shown by the hierarchy in Figure 4, subclasses inherit all public methods and public attributes from its super class. ArchetypeConstraint is the super class in the package and its two subclasses are CObject and CAttribute.

A list of the concrete objects with a short description follows; these objects correspond to nodes in the archetype definition tree (2 p. 11):

- CComplexObject: an object representing a constraint on instances of some non-primitive type, i.e. reference model classes like ENTRY, SECTION.
- CAttribute: an object representing a constraint on an attribute in an object type. An attribute is any data property of a class, the attribute can be both a relationship between classes and primitive attributes in the object, i.e. sting, integer etc.
- CPrimitiveObject: an object representing a constraint on a primitive object type, i.e. string, integer etc.
- ArchetypeInternalRef: an object that refers to a previously defined object in the same archetype.
- ConstraintRef: an object that refers to a constraint usually on a text or coded term entity that exists in the ontology section of the archetype.
- ArchetypeSlot: an object that defines a restriction on which other archetypes can appear at that point in the current archetype.

### 2.1.5 Archetype Definition Language (ADL)

ADL is a formal language for expressing archetypes and the ADL syntax is one possible serialization of an archetype. ADL uses three syntaxes, cADL (constraint form of ADL), dADL (data definition form of ADL), and a type of first-order predicate logic (FOPL), to describe constraints on data based on some information model (15 p. 13). When expressing an archetype in ADL the cADL syntax is used to express the archetype definition while the dADL syntax is used to express data in the sections language, description, ontology and revision history. The top-level structure of an ADL archetype is shown in Figure 5.

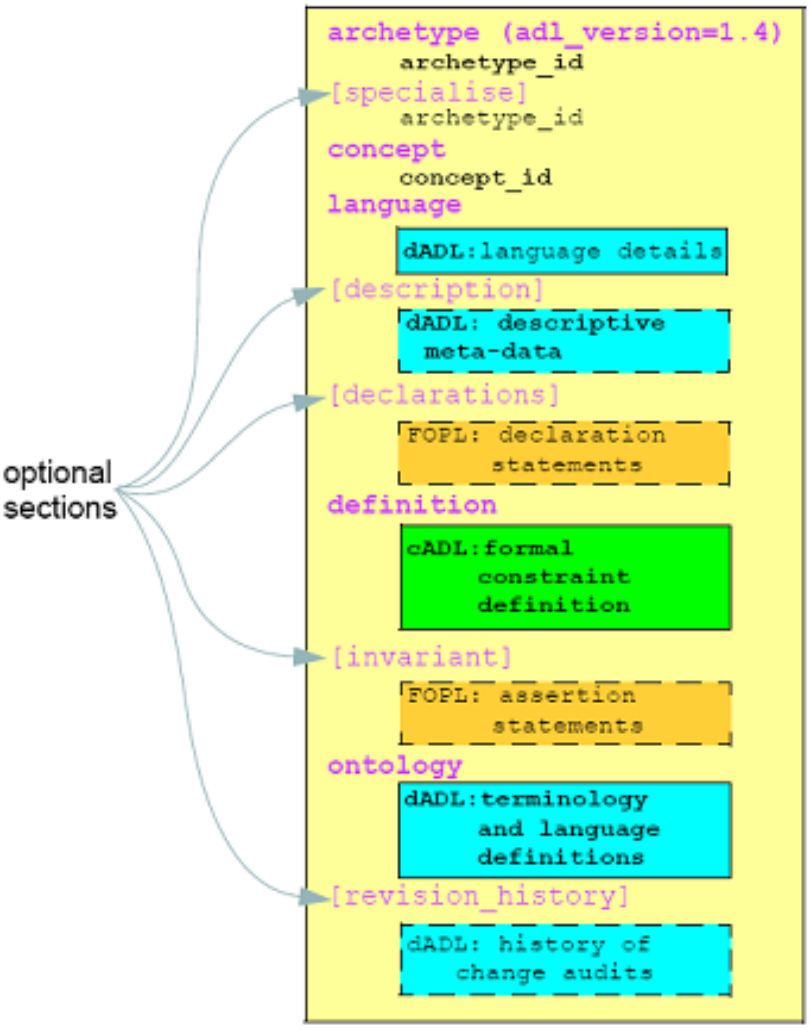


**Figure 5: ADL Archetype Structure (15 p. 12)**

The cADL syntax enables constraints on data defined by object-oriented information models to be expressed in for example archetypes. An example of how cADL may look is presented below. Comments in the code begin with "--".

```
PERSON[at0000] matches { -- constraint on PERSON instance
        name matches { -- constraint on PERSON.name
                TEXT matches {/.+/} -- any non-empty string
        }
        addresses cardinality matches {0..*} matches { -- constraint on
        ADDRESS matches { -- PERSON.addresses
                        -- etc --
                }
        }
}
```

The basic principle of dADL is to be able to represent instance data in a way that is both machine-processable and human readable (15 p. 23).

A common question is why dADL is used instead of XML. The origin of this question lies often in a widespread misconception of XML, that it is intended for humans because it can be read by a text editor (15 p. 23). XML is designed for machine processing and is textual for interoperability; this fact can be seen in realistic examples of XML (i.e. XML schema instances, OWL-RDF ontologies) that generally are unreadable for humans (15 p. 23). On the other hand dADL is intended to be human-writable and readable that is also machine processable. Some differences between XML and dADL are stated below but this is biased information from the creator of the ADL language (15 pp. 23-24):

- dADL provides a more comprehensive set of leaf data types (String, Integer, Date, Duration, etc.) compared to XML.
- dADL follows object-oriented semantics, particularly for container types, which XML schema languages usually don't.
- dADL isn't using XML notions of 'attributes' and 'elements' to represent what are object properties, this can create misunderstandings.
- dADL halves the space needed compared to an equivalent XML syntax.

It's good to have in mind that ADL is a language for archetypes and XML is a more general language. To make a scientific comparison of ADL and XML more research is needed but this is outside the scope of this thesis.

A common path syntax is used to reference nodes in both dADL and cADL. The same path syntax works for both since they have an alternating object/attribute structure. The general form of the path syntax is as follows (15 p. 85):

**path**: [‘/’] path_segment { ‘/’ path_segment }+

**path_segment**: attr_name [ ‘[’ object_id ‘]’ ]

ADL paths consist of segments separated by slashes (‘/’), where each segment is an attribute name with optional object identifier predicate, indicated by brackets (‘[]’). The path concept is illustrated in Figure 6 below.

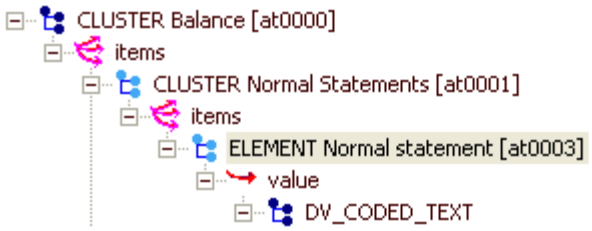


**Figure 6: Segment of an archetype where Balance [at0000] is the root node**

The path to the highlighted element [at0003] is: /items[at0001]/items[at0003].

# 2.2 Templates

This chapter describes templates, what they are and why they are needed.

## 2.2.1 What is a template?

OpenEHR templates (12 pp. 9-14) are closely related to openEHR archetypes. Templates are defined locally on the contrary to archetypes that define widely re-usable components of information. Templates also describe local usage of archetypes and relevant references. They modify the archetype concept with the following aspects (16):

- Archetype ‘chaining’: constructs larger structures consisting of multiple archetypes.
- Local optionality: possibility of narrowing the optional constraints (0...1) to either mandatory (1...1) or removal (0...0) for local needs.
- Tighten constraints: tightening constraints like cardinality, value ranges, terminology value sets, etc.
- Default values: default values can be specified for use in template structure at runtime.

### 2.2.2 Purpose of templates

Templates are used to limit archetypes to what they are intended to do for a particular application. For an examination of blood pressure of a pregnant woman for example, then the complete archetype of blood pressure is unnecessary. Instead the archetype is limited to the fields and allowed values actually needed for this examination.

## 2.3 Application Programming Interface (API)

This chapter describes what an API is, why API design is important and also some common design principles of APIs.

### 2.3.1 What is an API?

An API can be described as a set of standardized requests that have been designed to enable the developer to request services from the program (17). In what way the developers are to make the requests are described in the documentation, in this case in the Java documentation. Building an application without an API is a very bad idea since you would have a lot of trouble exchanging information in and out of the application.

If proprietary software is used the only way to access information is through APIs since the outside developers have no idea of how data are gathered and calculated. Of course it is important to have APIs even when using open source since it can be time-consuming to understand source code and misunderstandings can be avoided by using the standardized requests.

When designing a GUI (Graphical User Interface) it is very helpful if a good and complete API is used to minimize the need to get involved in the inner mechanisms of the source code. Instead the focus can be to make the GUI as useful as possible and when a certain feature is needed the correct API method is called without having to understand exactly how the method solves the problem.

### 2.3.2 The importance of API design

When developers choose to use a certain application it is important that the APIs are well designed because in many cases the API will be used for a long time. The developers often invest a lot of resources (money, time) creating their own application based on a number of APIs. If the API is solid it can be used for a long time and the customers can avoid having to stop using a certain API. A bad API design leads to either a lot of maintenance or that nobody uses it (18).

But what are the characteristics of a good API? According to the principal software engineer on Google Joshua Bloch the following list characterizes a good API (18):

- Easy to learn
- Easy to use, even without documentation
- Hard to misuse
- Easy to read and maintain code that uses it
- Sufficiently powerful to satisfy requirements
- Easy to extend
- Appropriate to audience

### 2.3.3 Design principles

The first step is to find out what the requirements are and these are most easily found out by thinking about the different use-cases that exists. To get feedback from actual users and the customer is very important to get correct use-cases. A good idea is to start making API calls even before the API functions are implemented, this can save time of finding out what functions are necessary (18).

The functionality of the API should be easily explained, even without documentation. Good names are important and if a certain function is hard to name it could be a bad sign meaning that the particular function isn't needed or it needs to be split up into a number of functions (18). All cryptic abbreviations of names should be avoided, standard naming conventions should be followed and the same word should always mean the same thing throughout the API (18). Even if the functionality is understood without the documentation, documentation should exist. All classes, interfaces, methods, constructors, parameters, exceptions and also state spaces should be documented because this is important to be able to reuse modules or parts of code later and of course to be able to easily maintain and improve the API (18).

Minimizing accessibility to classes and members by making them private is important to avoid confusion as much as possible. Implementation details of the API are not essential for a GUI developer to understand so they should be shown as little as possible (18).

## 2.4 The MVC design pattern

The Model-View-Controller design pattern is used to completely separate the GUI from the API. This makes modifications of either the GUI or the API easier since the developer only needs to have knowledge of one of them to make modifications. If one want to be able to support multiple types of clients it is necessary to separate core business model functionality from the presentation and control logic (19). Here the core business functionality represents the model and this is all the data access and the data processing. The presentation is the graphical part of the interface and the control logic is the user interaction with the different objects.

A simple example of where multiple clients might be necessary is for an online store. This example is cited from the document Java Blueprints Model-View-Controller (19). A HTML front for the web customers is required and for some of the wireless customers a WML front (Wireless Markup Language). The administrator of the online store would benefit from having a Java-based interface. The suppliers would probably want to use a XML-based web service. For all these different clients it is absolutely necessary to design with the MVC principles to avoid having to duplicate non-interface-specific code for each application, resulting in a lot more implementation-time, testing and maintenance (19). A diagram of the MVC pattern can be seen in Figure 7.

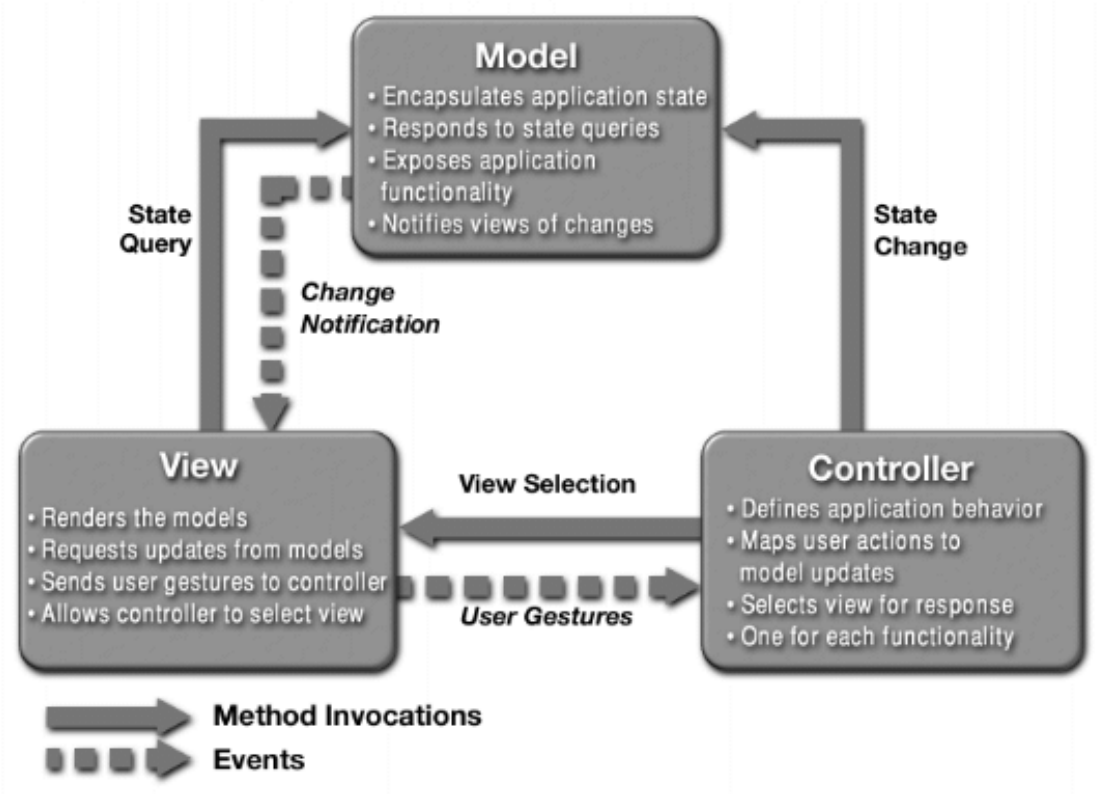


**Figure 7: The diagram represents the MVC (Model-View-Controller) pattern (19)**

## 2.5 License

The tri-license: The Mozilla Public License (MPL) (20), The General Public License (GPL) (21) and the Lesser Public License (LGPL) (22) is used in this project because the open source code that is utilized and all software copyrighted by the openEHR foundation uses this license.

## 2.6 Coding conventions

Naming conventions and code conventions are important for many reasons. They improve the readability and this is important in an API because it helps the users to understand the code.

When developing this API the code conventions dictated by Sun Microsystems has been followed as much as possible. For a complete reference read the document Code Conventions for the Java Programming Language (11).

The naming conventions can be summarized by the following list (23):

- packages
  i.e. DATA_TYPES.BASIC → datatypes.basic
- classes
  i.e. DATA_VALUE → DataValue
- fields
  i.e. calendar_alignment → calendarAlignment
- methods
  i.e. is_stricktly_comparable_to() → isStricktlyComparableTo()
- Accessors and mutators
  Fields (attributes) that are defined in the specification should be implemented as private fields with public accessors (getters), and mutators (setters) should provide access and possibility to manipulate them.

All code is commented with doc comment so that the Javadoc Tool can be used. For more information about how to write correct doc comments read the document How to Write Doc Comments for the Javadoc Tool (24).

The guideline criteria that have been followed in this project are to conserve the openEHR specifications that are described in the openEHR RM Java ITS document (23). The first two criteria are most important:

- The Java implementation must present the information that can be presented by the original class model in openEHR specifications.

- The implementation should have a similar look to the original model in terms of class names; attribute names etc, so a mapping between the original model and the implementation can be easily made.

## 2.7 Software verification and validation

Verification and validation (V&V) are two important steps for domain software quality. Software V&V is a disciplined way of evaluating software products throughout the products lifecycle (25). A V&V based development strives to ensure that quality is built into the system and that the software fulfills the requirements. V&V have become very important in software as the complexity of software systems has increased and the planning of V&V is necessary from the beginning of the development life cycle (25) (26) (27). This is significant for the openEHR archetypes, not only for their complexity but also the medical domain demands additional caution when it comes to quality.

According to the IEEE Standard Glossary of Software Engineering Terminology verification is “The process of evaluating a system or component to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase“ (27). Software verification techniques can be broken down in two categories: dynamic testing and static testing (27). The dynamic tests require the execution of software and can be further divided into three groups: functional testing, structural testing and random testing (27) (28).

Functional testing involves identifying and testing all the functions of the system in order for them to meet their requirements. This type of testing is an example of black box testing. Black box testing means that the tester creates test cases that are either functional or non-functional and apply them to the testing object. In black box testing there is no knowledge of the test objects internal structure.

Structural testing refers to a testing form where the tester has full knowledge of the implementation of the system and is therefore an example of white-box testing (27). In structural testing the tester uses information from the internal structure of the system to form tests that check the operation of individual components (27).

Random testing is when the tester freely chooses test cases among all possible test cases. Under this form of testing, one can classify for instance exhaust testing, where the tester tries all possible input values in a function (27).

Static testing includes testing that does not involve the execution of the software at test. The static testing techniques rely on the manual examination (reviews) and automated analysis (static analysis) of the code (28). The manual examination is an ongoing testing during the whole development cycle and can be performed before dynamic tests. The static analysis is done with tools that analyze program code i.e. the compiler in Java. Static testing finds defects rather than failures, where defects refer to nonfulfilment of intended usage requirements and failures denotes “deviation of the delivered service from compliance with the specification” (29).

Validation usually takes place at the end of a development cycle and concerns primarily the complete system as opposed to the verification that focuses on smaller sub-systems. The IEEE Standard Glossary of Software Engineering Terminology defines validation to be “The process of evaluating a system or component during or at the end of the development process to determine whether it satisfies specified requirements” (27). Software validation is dependent on comprehensive testing, inspections, analyses and other verification tasks so V&V partly overlap when it comes to techniques.

## 2.8 Background to the validation

This chapter presents some background information to understand how the validation of archetypes work. The validation that was used in this project is an implementation of the LinkEHR validation. The source code used was taken from a zip-archive downloaded from the official homepage of LinkEHR (30).

The validation methodology is inspired by the document “Taxonomy of XML schema languages using formal language theory” (31) which presents validation for XML schemas. Since it is relevant for understanding the validation algorithm a short introduction to the area is necessary. This introduction is rough and (31) should be read to get a deeper understanding of the area.

XML documents can be presented as trees rather than strings, which is why tree grammar can be used to express XML content. A tree grammar is a formal grammar that allows generating trees (31). In this context regular trees are defined as ordered (a node has an ordered sequence of child nodes) and a node is not allowed to have any number of child nodes. All nodes are also labeled, with the exception of text nodes that are leaves. These kinds of trees capture the element structure of XML documents (31).

A regular tree grammar is a 4-tuple G = (N, T, S, P) where: N is a finite set of non-terminals, T is a finite set of terminal symbols, S is a set of start symbols

and S ∈ N, P is the set of production rules of the form $X \rightarrow \mathbf{a}\ r$ (e.g. *Doc* → **doc**(*Para1*, *Para2*)), where $X \in$ N, **a** ∈ T, and $r$ is a regular expression expressing the content model of this production rule. $r$ consists of terms that are part of N, $r \in$ N. The terminal symbols are represented with **bold** lowercase and non-terminal symbols are represented with capitalized *Italic*. The null sequence of non-terminals is represented by ε.

An example of a regular tree grammar is presented below.

G = (N, T, S, P)

*N* = {*Doc*, *Para1*, *Para2*, *Pcdata*}

*T* = {**doc**, **para**, **pcdata**}

*S* = {*Doc*}

*P* = {*Doc* → **doc**(*Para1*, *Para2*), *Para1* → **para** (ε),

*Para2* → **para** (*Pcdata*), *Pcdata* → **pcdata** (ε)}

This example spans the following tree according to the regular tree grammar.

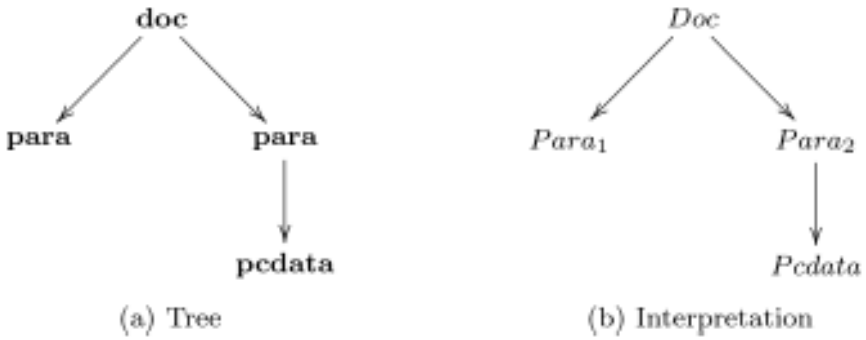


**Figure 8: The example tree t and its interpretation against G**

Figure 8 is an interpretation *I* of a tree t against a regular tree grammar G where *I* is a mapping from each node *e* in t to a non-terminal. The interpretation is denoted *I(e)* such as:

1. *I(e)* is a start symbol when e is the root of t, and
2. for each node e and its child nodes $e_0, e_1 \ldots, e_m$, there exists a production rule $X \rightarrow \mathbf{a}\ r$ in G such that
3. *I(e)* is *X*
4. the terminal symbol (label) of **e** is **a**, and
5. $I(e_0)\ I(e_1) \ldots I(e_m)$ matches $r$

A tree is valid against a regular tree grammar G if there is an interpretation of t against G. This concept is extended in (31) for two other classes of tree grammar called local and single-type.

One of the validation algorithms (31) that validates regular tree grammars is presented in Figure 9.

**Algorithm 3:** Validation for `regular tree grammars`.

**Input** : XML document $D$

Let $S$ be an empty stack of lists of sets of nonterminals;
//Note that we have to use *sets* of nonterminals rather than nonterminals.
Let $Y$ be an empty stack of *sets* of production rules;
//Note that we have to use *sets* of production rules rather than
//production rules.
Let $P$ be an empty stack of nonterminal sets;
Push the set of start symbols into $P$;
**traverse** $D$ **in the depth-first manner**
  **when** *element* $e$ **is visited**
    find production rules of the form $X^i \rightarrow \mathbf{a}\ (r^i)$ such that $\mathbf{a}$ is the tag name of $e$ and $X^i$ is contained in the nonterminal set at the top of $P$;
    //More than one such production rule may be found.
    //$X^i$ is an applicable nonterminal.
    **if** *no such production rule is found* **then**
      report "invalid" and halt;
    push $\{X^i \rightarrow \mathbf{a}\ (r^i) \mid i = 1, 2, \ldots\}$ to $Y$;
    push an empty list to $S$;
    push the set of nonterminals occurring in some $r^i$ into $P$;
  **when** *element* $e$ **is exited from**
    pop $\{X^i \rightarrow \mathbf{a}\ (r^i) \mid i = 1, 2, \ldots\}$ out of $Y$;
    pop a list of sets of nonterminals $(\mathbf{X}_1, \mathbf{X}_2, \ldots, \mathbf{X}_n)$ out of $S$;
    //$\mathbf{X}_1, \mathbf{X}_2, \ldots, \mathbf{X}_n$ are *sets* of nonterminals assigned to
    //the children of $e$;
    let $\mathbf{X}$ be the set of $X^i$ such that $M[r^i]$ accepts $(\mathbf{X}_1, \mathbf{X}_2, \ldots, \mathbf{X}_n)$;
    **if** $\mathbf{X}$ *is empty* **then**
      report "invalid" and halt;
    append $\mathbf{X}$ to the list of sets of nonterminals at the top of $S$;
    pop a nonterminal set out of $P$;
report "valid";

**Figure 9: Algorithm for validating regular tree grammar (31).**

The logics presented in (31) could be projected on validation of the definition part of an archetype since the structure of the definition section is similar to trees generated by regular tree grammar. Therefore LinkEHR have made a generalization of the algorithm in Figure 9 to function on an archetype's definition section. The LinkEHR principles for validation are presented in Framework for clinical data standardization based on archetypes (32 pp. 454-458) and the validation algorithm can be found in appendix 8.3.

## 2.9 Data model for validation

In order for the mentioned validation algorithm for archetypes to work some modifications has to be done in the data model for representing archetypes. These modifications are done to make the representation of data instances more straightforward and formal to be more compatible with a regular tree model (32 pp. 454-458). In this data model each object is described by a data tree where the root node is labeled with the class name and has one child for each attribute (32 pp. 454-458). Furthermore the children are labeled with the attribute names and each of them has one child labeled with the corresponding type (class) name. This mechanism is repeated iteratively in the model and atomic values are represented by a leaf node labeled with a value. The data model is slightly different from the AOM 1.0.1 and the difference lay mainly in the representation of leaf and near leaf objects. An example of how these differences may look is presented in Figure 10. The figure shows how the data is remodeled to be compatible with the validation of LinkEHR. The first object (at0007) in the list of at-codes to the left is expanded in the representation to the right and the at-code at0007 can be found in the upper CPrimitive object String.

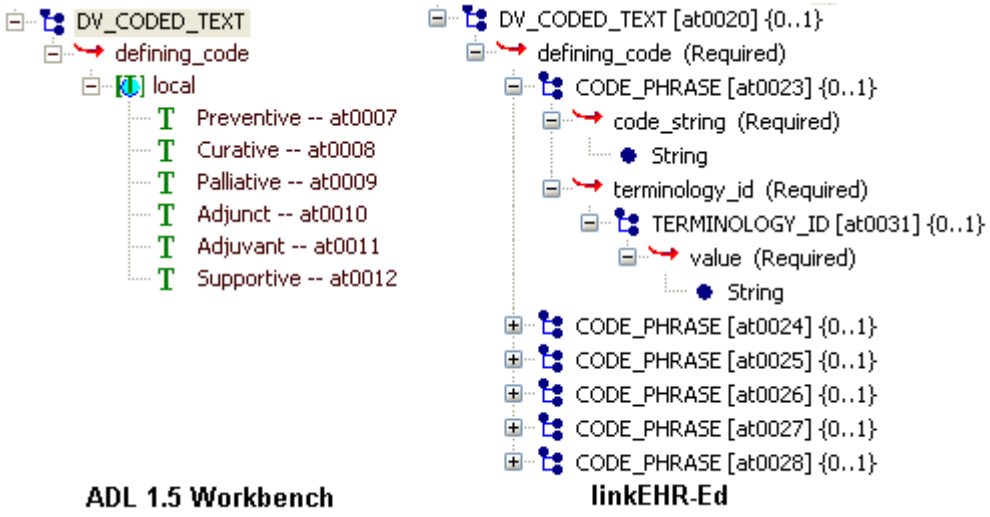


**Figure 10: Representation of a CodePhrase in workbench and LinkEHR-Ed, where ADL workbench is representing the tree according to AOM 1.0.1**

## 2.10 Subsumption used in validation

Subsumption is used in the validation; this chapter shortly describes the term subsumption based on (33). This section will introduce the subsumption idea on a database but the logic can be applied to the archetype domain. The

database in this model is designed to be a structure D consisting of a set of object ids $O_D$, a fixed set of labels denoted $label_D$ and children denoted $children_D$. Lets introduce T to be a fixed set of type names and elements of T are $\tau$, $\tau'$ etc, which can be related to the object id. The database is a tree with a root denoted as $\Delta$. Briefly, subsumption relies on a mapping between types and on inclusion between children over these types.

A definition from (33) states the following:
**Definition**. Let S and S′ be two schemas and S subsumes S′ under the subsumption mapping $\theta$. The subsumption $S \leq_\theta S'$ exists if $\theta$ is a function from $T_S \cup \Delta$ to $T_{S'} \cup \Delta$ such that:

1. $\theta(\tau) = \Delta$ if $\tau = \Delta$.
2. For all $\tau \in T_S$, $label_S(\tau) \subseteq label_{S'}(\theta(\tau))$.
3. For all $\tau \in T_S \cup \Delta$, $\theta(L(children_S(\tau))) \subseteq L(children_{S'}(\theta(\tau)))$. Where L is a fixed set of labels.

Subsumption mapping is presented in Figure 11 as the step from the 2nd layer to the 3rd, while the 1st layer to the 2nd describes a type assignment that is irrelevant in this context.

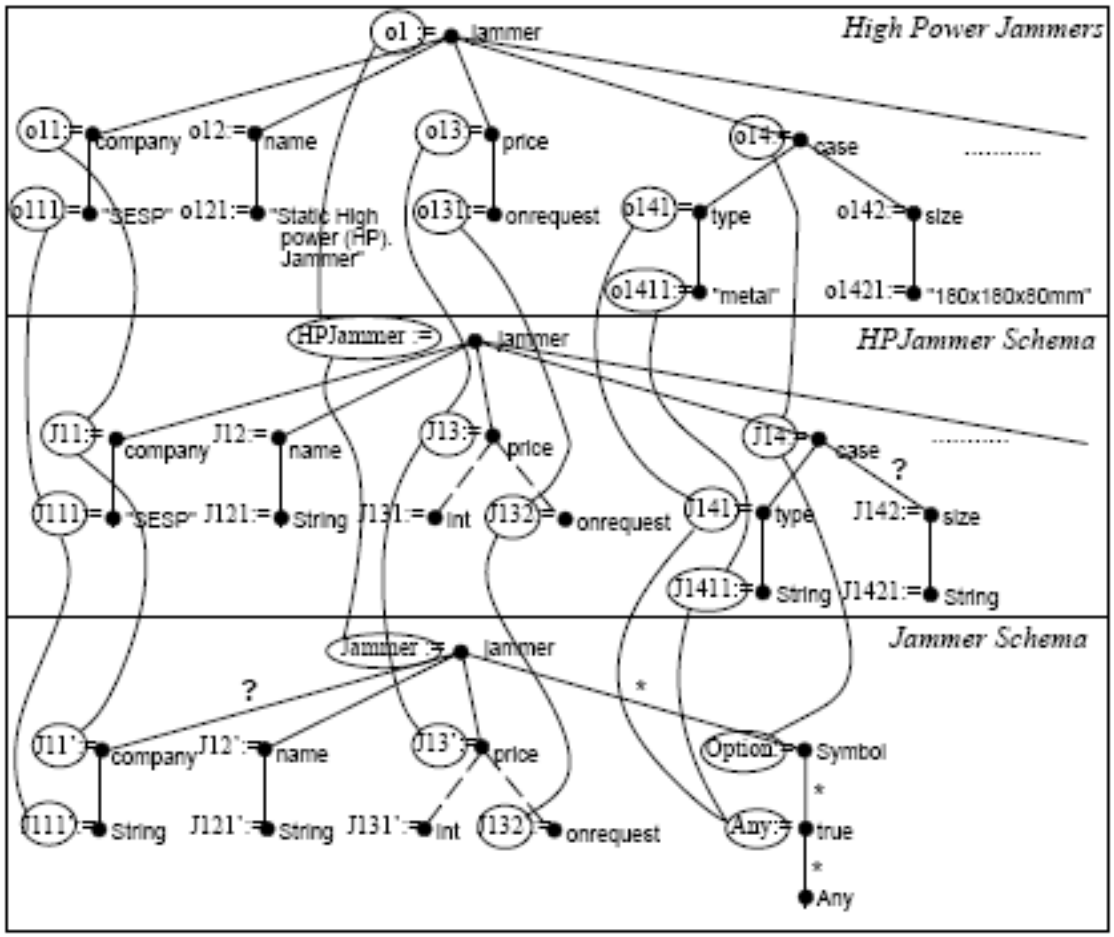


**Figure 11: Type assignment (1st to 2nd layer) and subsumption mapping (2nd to 3rd layer)**

Figure 11 illustrates the subsumption mapping between Jammer and HP Jammer types, corresponding to the following θ′:

$\theta'(HPJammer) = Jammer$ $\theta'(J_{11}) = J'_{11}$

$\theta'(J_{111}) = J_{111}$ $\theta'(J_{13}) = J'_{13}$

$\theta'(J_{14}) = Option$ $\theta'(J_{141}) = Any…$

### 2.10.1 The subsumption theory used to validate archetypes

A subsumption function is a set of mappings for the child archetype to the parent archetype. Each mapping goes from an attribute or object to another attribute or object (34). A mapping from X to Y says that the attribute or object Y is more (or equally) restricted than X. An archetype B is more general than an archetype A, when it is possible to find such mapping for every attribute and object of B (34). The method testContainment, which is called in the implemented validation method, looks for a subsumption function. It goes bottom up, so it first check the domain of atomic attributes (for instance the containment of intervals or the containment of regular expression) and then goes up until the root node.

## 2.11 Mutable and immutable objects

A mutable object is an object that is capable of change, that is, it can be changed after creation. Immutable objects are the opposite of this which means they can’t change after creation.

If an archetype is created using an immutable model it means that all the different parts of the archetype have to be included at creation because once the archetype is created it can’t be changed. The only way to edit the archetype after creation is to create a new archetype object with the wanted changes. When a mutable model is used changes can be done after creation.

There are benefits of making objects immutable e.g. if the object is to be sent between different processes in a computer program it is recommended that the object is made immutable because it will always contain the same information. Even if ill-behaved code tries to change the state of the object it won’t be possible and this creates higher level of security (35).

It is generally better to use immutable objects for smaller objects which are mostly used for transporting information and only need a few or no changes, e.g. abstract datatypes (35).

For large structures (e.g. CComplexObject) that often change it is inefficient to use immutable objects since the whole structure would have to be recreated every time a new change is made (35).

The whole structure is however not required to be mutable or immutable. Developers can choose to make parts of the structure immutable and other parts mutable. This has been done in the API of this project. Smaller objects are designed to be immutable while other objects that are in need of more changes are designed to be mutable.

# 3 Method

This chapter describes the developing method used in the master thesis project.

## 3.1 Development process

Before the master thesis proposal the plan was to develop the existing Liu-editor and make it more understandable and better documented. There were also thoughts of making the master thesis project based on developing validation mechanisms for archetypes. After about 4 weeks the focus landed on making an API for creating and editing openEHR archetypes and the requirements that are stated in the first chapter was derived shortly after and a master thesis proposal could be presented around 6 weeks after the start.

The development process started with reading the openEHR specification with the main focus on the AOM document. Parallel to this time was spent on understanding and learning more about the existing archetype editors/viewers. The openEHR Java implementation project was also studied in the early stages of the master thesis project. A design plan for using this implementation and develop it further to fulfill the requirements was created. A decision was made to integrate the validation functionality of LinkEHR since this was the only seemingly working semantic validator available except ADL workbench which is written in the Eiffel language (36) and to make a port from Eiffel to Java would require more time than available for this master thesis project.

Most of the coding took place during the implementation phase where the mutability functionality was implemented first. After that the development continued with writing new methods and at the same time tests for these methods. Furthermore the development continued with integration of existing features like the ADL-parser, validation and serializer. Documentation was performed in the form of documentation for Java, logs for keeping track of changes to the original Java implementation and more detailed descriptions in the master thesis report. In the end stages of this phase some design thoughts on how a possible GUI could be implemented was formed.

The end phase included fine-tuning and finalization of the master thesis document and code. During this phase time was spent on understanding and writing about the LinkEHR validation more and give a GUI proposal. Some more features for the API where also finished in this phase, for example the ability for the user to set immutability.

## 3.2 Testing

This section describes the testing process of the project.

### 3.2.1 Unit testing

A unit is the smallest testable part of an application and in the API a unit refers to an object method. The goal of unit testing is to isolate each part of the API and show that they work properly. Unit tests are constructed as strict written contracts, usually common comparison and condition testing that the unit must satisfy in order to pass the test. A great benefit with unit tests is that one can test parts of the program early and separate different methods from each other. This helps because problems are found early in the development cycle. Unit tests are used to see that the behaviors of an application are not changed when modifications are made to the source code (37).

In this project JUnit 4 was used. JUnit is the informal standard when it comes to unit testing in Java (38). There are some other testing tools that have risen in popularity the last years but after the release of JUnit 4 it has reclaimed the attention and has become the de facto standard (38). The fact that JUnit is well known and simple is very important since the API is supposed to be a foundation for future developed archetype editors. The plan is that the unit tests should be reused.

The design of the unit tests in the API is to implement one JUnit test case for each class that is tested. The aspects that should be tested in the unit tests are described in the following list:

- Negative tests to be sure that the method responds to error conditions.
- Test if the method behaves appropriately when giving invalid or unexpected input values.
- Test how methods work together by stacked tests that examine more complex behavior, for instance unit tests with many methods that interact with different classes.
- Test that methods work properly with correct arguments.

The coverage of the tests are based on which methods are tested e.g. simple set and get methods will not be as thoroughly tested as methods that includes many operations. Methods that change the archetype definition and ontology are tested more extensively than other methods since these parts are more complex than other parts of the archetype.

The unit tests for testing a specific method in a super class like the archetype class may also include methods from its subclass so many tests span over many levels of methods. When testing the setOntology method the tests include the creation of the smallest elements of an ontology such as an

ArchetypeTerm or a TermDefinition. Hence the test of super classes tests more than just the method itself.

#### 3.2.1.1 Class testing

The JUnit tests are built to either test separate methods of a class or the whole class at the same time. A class is considered to work correctly when all the unit tests of its methods passes.

### 3.2.2 Round trip test

Round trip tests will be performed on the API to test that the API behaves properly on a macroscopic level. These tests can have the following structure: Load an archetype with the ADL-parser, edit the archetype and serialize the archetype to ADL. After these steps the test finishes with an ADL syntax check, a comparison between the two ADL files to verify that the API has altered the correct information and that the API did not change something by mistake.

## 3.3 Development environment

The API is written in Java with the help from the Eclipse development environment. Java was chosen because of its platform independence.

# 4 Result

The API is built on the kernel from the openEHR Java reference implementation project but with a more mutable AM. Additional features have been added to the API, for instance an ADL-parser, an ADL-serializer and a validator. The API is, as requested, Java based which makes it platform independent. The API is licensed under the Mozilla Public License. The API is built to be compatible with the AOM specifications 1.0.1 which make the system more “future friendly” and easier to maintain. All new methods have a declaring Java documentation to provide a clear picture of each method.

When implementing the API based on the openEHR Java reference implementation project classes have been added one at the time to minimize the possibility of problems. This means that the API as it is right now only includes classes used by the API methods. If a class that does not exist is needed in the future it needs to be copied from the openEHR Java reference implementation project. This is something that will be required to do before creating a GUI application with this API.

In Figure 12 you can see an UML diagram of the constraint_model package of this project. Something to notice is the difference to the original source code, see Figure 15 in appendix 8.4.1. The inheritance changes that can be seen were made to follow AOM 1.0.1, which the original source code didn’t. Notice also that new methods have been added to most classes of the constraint_model package. These changes were made to integrate validation procedures and to add more functionality for editing archetypes.

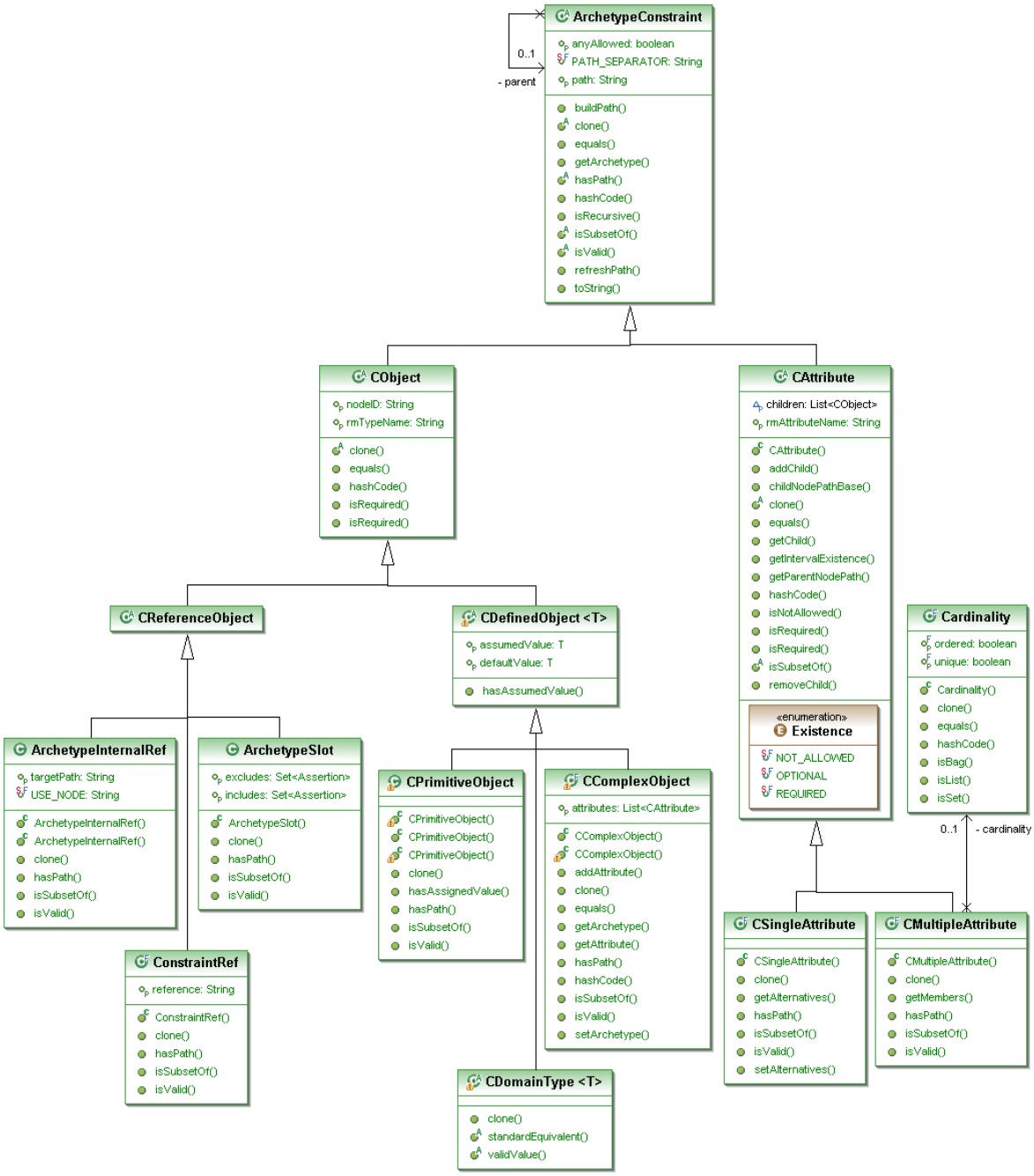


**Figure 12: An UML diagram of the classes in the constraint_model package of our project.**

For comparison the UML diagram of the openEHR Java reference implementation project can be seen in Figure 15 in chapter 8.4.1. Something that is not shown in this figure is that ArchetypeConstraint inherits from the AMObject class and that is not the case for the Java reference implementation project. The reason the ArchetypeConstraint class inherits from AMObject is because the immutability setting must be in common for all objects that are part of the archetype, otherwise all objects are not immutable when the setting is active.

## 4.1 Functionality of the developed API

### 4.1.1 Archetype class

The central class of the API is the Archetype class. This class contains all methods that can be applied to an object that is defined as an archetype. This section that describes the Archetype class is divided into three abstract parts: description, definition and ontology. For a more detailed description of the implemented methods see appendix 8.1.

#### 4.1.1.1 Description

The description part of the archetype contains methods that mainly manage meta-information. Examples of methods that are included in this domain are get and set methods for conceptcode, archetype-id and ADL version. This part of the archetype class is quite similar to the archetype class from the openEHR Java reference implementation project.

#### 4.1.1.2 Definition

This segment of the class manages the actual definition of the archetype. Here the main methods are get- and setDefinition, where the argument is an object of type CComplexObject. This part of the archetype class is where the most development has been made. Some methods that have been implemented are removeCObjectAtPath, correctParentsInArchtype and ValidateArchetype.

#### 4.1.1.3 Ontology

The ontology is the part of the class that manages the semantics of the archetype. The main methods would be get- and setOntology, which is of the class ArchetypeOntology. Some methods that have been implemented are deleteTermDefinition and deleteTermBinding.

### 4.1.2 Optional immutability

A Boolean field named *immutable* have been added to the RMObject class with false as default value. This field has a method setImmutable which sets the field to true and this means that the object is immutable. The RMObject class is a parent class to the AMObject class which is a parent class to the ArchetypeConstraint class. The ArchetypeConstraint class is the parent class to all objects in the constraint model package, see Figure 4. When the field *immutable* is true all classes inheriting from the RMObject class are read-only (immutable). This is enforced by the method assertMutable which runs every time before a value can be changed. The method assertMutable throws an ImmutableException if the field *immutable* is true and therefore preventing changes.

### 4.1.3 Contributions to the openEHR Java reference implementation project

During the project a number of faults were found in the openEHR Java reference implementation project and most of those were reported right away and fixed shortly after. Some of the faults are listed in this chapter.

A TerminologyService problem was found regarding the codeset name “languages”, which was hard-coded instead of using a static field. This caused the terminology service of the TranslationDetails class to throw an exception because “languages” was not an externalId field in the XML terminology file.

There were two inheritance inconsistencies to the AOM specifications 1.0.1 in the constraint_model package. The two abstract classes CReferenceObject and CDefinedObject were implemented but none of them were used. Instead some of the concrete classes (ArchetypeInternalRef, ConstraintRef, ArchetypeSlot, CPrimitiveObject and CComplexObject) and the abstract class CDomainType inherited directly from CObject. This forced the concrete classes to be larger than necessary by populating them with fields and methods that should have been inherited from CReferenceObject and CDefinedObject. This was fixed by changing the inheritance to the AOM 1.0.1 specification, see Figure 4.

The constructor of the class CDvOrdinal in the package openehrprofile.datatypes.quantity has two fields, defaultValue and assumedValue. defaultValue was of type CDvOrdinal and assumedValue of type Ordinal. This was not according to AOM 1.0.1, though it wasn’t a big problem since “Ordinal is really a mid-way solution which only keeps the essentials from the constraint” according to Rong Chen (39). This change was however necessary for the LinkEHR validator so it was changed and at the same time the change was according to the AOM 1.0.1 so it was necessary.

### 4.1.4 Verification & Validation

The validator implemented in the API was taken from the validator from the open source project LinkEHR (40). The validator class can be found in the folder org.upv.ibime.linkehr.semantic in the source to the main folder Archetype API. The main method in the validator class is testContainment whose purpose is to test subsumption between a parent archetype and its child. The openEHR RM classes are also treated as archetypes, which make it possible to validate an archetype with respect to its RM type. The intention of the testContainment method is to verify if the parent defines broader constraints, i.e. has more general or identical constraints as the child. The algorithm in the method takes two archetypes as arguments and calculates the set of subsumption functions between them (34). The validation uses a generalization of the algorithm presented in Figure 9.

The inheritance relationship between archetypes is modeled with a subsumption relation (34). This subsumption captures both the containment relation between two archetypes and also the structural relationships between node objects from both archetypes. The latter means that the subsumption mappings specify specialization relationships among nodes of the child and the parent archetype. This feature is compatible with the syntactical rules that ADL uses to specialize nodes. For more details regarding the validation read “Framework for clinical data standardization based on archetypes” (32 pp. 454-458).

The implemented method ValidArchetype can only validate archetypes of the type ItemTree. This is because only one of the RM files that are used as a validation reference (i.e. used as a parent archetype in testContainment) has been updated to pass the new ADL parser, made by Acode.

All that has to be done to make validation of other files possible is to update the RM archetypes files to pass the ADL parser. To do this it helps to run ADL workbench to get helpful error messages. One way to correct these RM files is to systematically correct the flaws relating to the error messages from running an ADL workbench validation.

The kernel itself provides some controlling features by throwing exceptions if something passed to a method is wrong. These checks mainly concern the constructors of the archetype model classes and small entities. All these verifications help the programmer to build archetypes from the smallest parts to huge structures in a more correct fashion.

The ADL-parser verifies that the syntax in the ADL file is correct and then converts it into a Java object. An overview of the parsing process can be seen in Figure 2.

### 4.1.5 Changes made to implement the LinkEHR validator

To incorporate the LinkEHR validator into the original source code a lot of changes had to be made.

A field *parent* was added to the ArchetypeConstraint class as well as a method called isRecursive. Most classes associated with some part of the definition section of the archetype have been given a clone method and new constructors have been added to some classes, for example the CComplexObject class. A method that has been implemented partly for validation and partly as a help-method for other methods is correctParentsInArchetype.

## 4.2 Testing

A testing folder called “Archetype API junittest” exists which can be used to confirm that the methods in the API work. There are new JUnit tests that verify the new methods and there are old JUnit tests created by the openEHR Java reference implementation project to test that the changes that are made doesn’t change the functionality of the original Java implementation. The new unit tests use JUnit 4 and the reused test from the openEHR Java reference implementation project uses JUnit 3 but since there is compatibility between the versions there is no problem. Test suites are made to run all the tests within a folder (i.e. test folder for constraintmodel) so the tester doesn’t have to do the tests manually. The test cases have the same name as the class under test with the suffix Test e.g. CObjectTest.

The file ArchetypeTest.Java includes not only tests for the Archetype class methods but also a test to create an archetype using the full constructor to verify that small entities in building an archetype works. The archetype that is constructed includes all the concrete classes that an archetype can consist of in order to achieve a good testing object. This archetype wasn’t only created to be a testing object but also to ensure that the API could build more complex structures.

A test run for the Archetype class are shown in Figure 13. Worth mentioning is that this test doesn’t only test the Archetype class but instead this test span over the whole architecture but it only test small parts of other classes. An example is the test for get- and setDefinition that creates a definition with the constructors that build the smallest archetype constraints like CString to more complex archetype constraints like CComplexObject. This doesn’t mean that the ArchetypeTest can replace other unit tests but it means that many tests overlap and are broader than they seem.

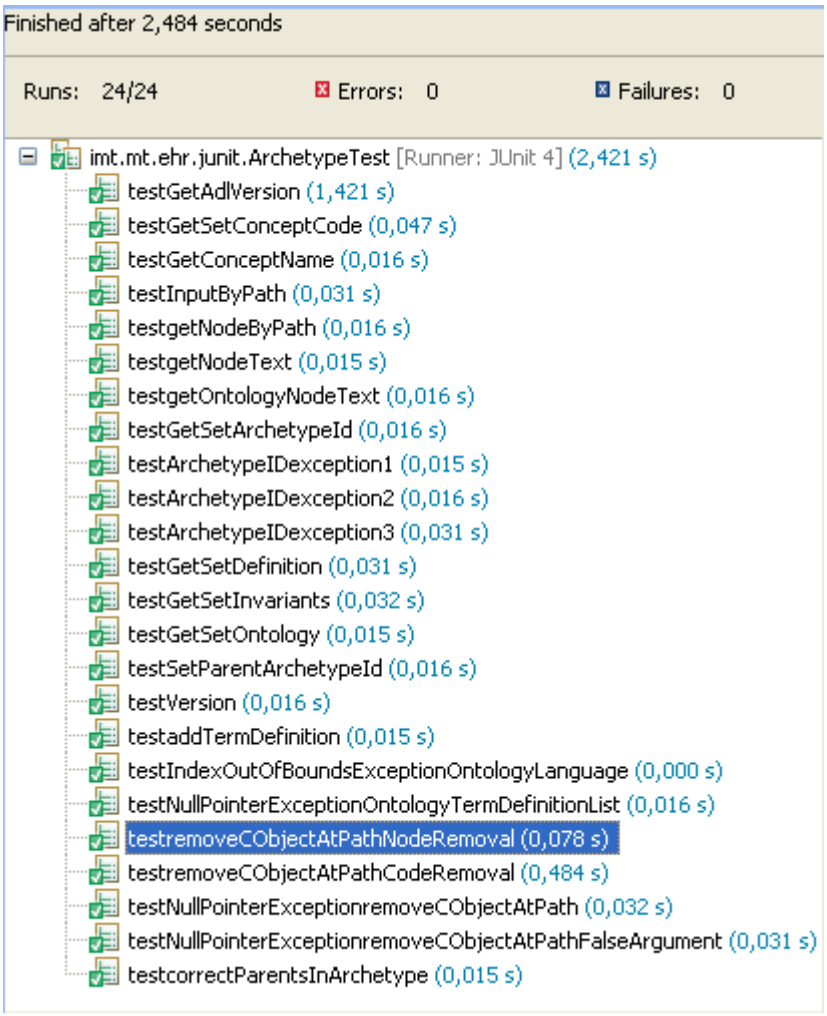


**Figure 13: JUnit test results for ArchetypeTest.class**

Figure 13 doesn't actually show that the tests prove anything or verify what is intended for them, a more detailed example of how the tests are written are shown in appendix 8.2 and also in the source code (41). The tests that are written in the testing folder can be used in future developments of this API or other related Java projects.

## 4.3 Help methods

In addition to the actual API a number of helpful methods have been created during the whole development process. These methods were created for learning purposes and some of them have been religiously documented to pass the knowledge on to whoever needs it in the future.

Most of the helpful methods are placed in the eclipse project named Archetype API test (41).

### 4.3.1 Batch validation

In BatchValidation.java there exist methods for batch validation of archetypes, in other words how to validate many archetypes in a row and save the results of each validation to a text file.

### 4.3.2 Archetype creation

In ArchetypeCreation.java it is shown how to create a correct archetype using the full constructor. This works for both our project and the original openEHR Java reference implementation project. It is described how strings and other arguments are supposed to be formatted to create a correct archetype. The creation process is divided into smaller parts by separate methods for most of the smaller objects of which an archetype consists. This means that you can easily create only a small object, for example an Assertion object (assertion enables constraints to be expressed in a structural fashion) or large object as the Definition object (root node of an archetype).

# 5 Discussion

## 5.1 Design decisions

The API has most of the methods that are needed for implementing a GUI, the API can build and edit archetypes, as can be seen in the tests. In this section some design decisions are discussed.

### 5.1.1 Description section

Since the description part of the archetype mainly concerns getting and setting strings there haven't been any major changes in this part. Some methods have been added but most of them remain unimplemented for now, these methods are added to follow the AOM specifications 1.0.1. The reason for adding these methods is, besides their addition to the API, to prepare and make the API "future friendly" if developers expect the system to be compatible with the specifications.

### 5.1.2 Definition section

One of the requirements of this project was to make the object model mutable, which makes it easier when creating archetype editors since the archetype object doesn't have to be recreated when changes are made. A key function when editing an archetype is getNodeAtPath, which returns the CObject at the given path. This function in combination with the mutable object model makes it easy to edit the definition part of the archetype. Traversing down the definition tree without getNodeAtPath is time-consuming and requires the user to know the structure of the tree or the names of the nodes.

The method getNodeAtPath is also used in the method removeCObjectAtPath that basically exists to automate all the actions that needs to be done when deleting a CObject. The fact that the method removeCObjectAtPath also deletes all unique at and at/ac-codes (ac stands for archetype constraint) from the ontology is very convenient. This means that the GUI programmer does not have to manually delete at/ac-codes when deleting nodes in the tree. This feature helps when deleting nodes that represent huge branches of the tree with assumingly many unique at/ac-codes.

Instead of completely removing the principle of immutability the API lets the user decide when to make the archetype object immutable. This is done by a Boolean field *immutable* in the RMObject class. This field has false as default

value which makes the archetype mutable, but when this Boolean is set to true it is no longer possible to change any data in the archetype. When the user tries to change some value an ImmutableException is thrown and this exception needs to be handled by the user of the API. This is however just superficial immutability and not true immutability since the fields are not final. Some methods have been left without the mutability check and this is because the validation procedure from LinkEHR needs to make some changes to the archetype to be able to validate it.

### 5.1.3 Ontology section

There is a field called *language* in the archetype ontology class. This field is not defined in the AOM 1.0.1 specification but we can still see a use for this field. The field language should not be confused with the language field in the archetype class, which sets the default language for the archetype. We assume that this field is supposed to be used as the default language for editing ontology and this is necessary since these two languages are not necessarily the same.

The API now has more basic methods to edit the ontology. These methods can be combined into helpful methods that can be used by a GUI developer. One example of such a method is a method that creates an archetype term with the active ontology language. Of course one can add a new language as well. With this language the method should construct an archetype term that takes three arguments: at-code, description and text. The archetype term together with the active language can construct an ontology definition that is the class that can be added in the archetype ontology lists like the list of term definitions.

An important understanding for a GUI developer is to connect the correct objects to be shown in the GUI. When it comes to the list of at-codes we recommend using the hash-list in the ArchetypeOntology because the key for this list is *language* and will be built in a more proper manner than for instance the term definition list, which is also found in the ArchetypeOntology. Generally the GUI presents all at-codes referring to the default archetype language and since the hash-map is ordered after languages it is the best solution for this task.

The openEHR Java reference implementation project immutable kernel doesn’t have any update calls for hash-maps except from when the archetype is loaded. After the modifications to make the kernel more mutable, updates for different maps were introduced. The modifications regarding this are done to all methods that alter the ontology lists. The changes that were done consist of overriding the add, clear and remove methods that concerns lists so they update the hash-maps when they are used.

### 5.1.4 Importance of following the specifications

It is important to follow the specifications. In the case of openEHR it is important that the new software constructs and handles an archetype in the specified way. It is also important from a compatibility point of view that future additions or updates will be compatible with the software. Therefore, one should never delete or change any existing contents; instead one should add features for extended functionality.

In the developed API there are many additional methods and some new fields but nothing has been deleted from the original implementation. When adding new methods or fields it is important to make sure they don't interfere with the specifications. LinkEHR for instance has a field *parent* in the class ArchetypeConstraint. They have this to be able to easily traverse the archetype structure, which is needed for their validation methods. This field was not a part of the AOM 1.0.1 specifications so when using the ADL-parser to translate ADL into an object tree this field is left empty. LinkEHR therefore changed the ADL-parser to fill in the *parent* field during parsing. This created maintenance problems since this means that for every new version of the ADL-parser these changes have to be made for the validation to work. Instead we created a new method called correctParentsInArchetype which is called after the ADL-parser; this method traverses the whole definition part of the archetype and fills the *parent* field for every ArchetypeConstraint object. With this method it means that a new version of the ADL-parser can easily be integrated in the future without changing anything which makes this a more modular solution than before.

## 5.2 Validation

There are two validation procedures with semantic validation functionality that we know of, ADL Workbench and LinkEHR. The reason we chose to use LinkEHR was that it was implemented in Java and there was not enough time to port ADL Workbench from Eiffel to Java.

The validation that was used in this project is an implementation of the LinkEHR validation. The source code used was taken from a zip-archive downloaded from the official homepage of LinkEHR (30). However this source code was not the latest revision since the released compiled version has other functionality. This was not known until late in the development cycle which has affected our project negatively. We assumed that the compiled version that was released was a compilation of the source code released on their official webpage but that was not the case. Since we were only able to compile one module at the time we were unable to create an

executable version. Therefore the released compiled version was used for the basis of how their application worked. Later in the project some important features didn't work as expected, for example the validator which had a hard time accepting any archetype at all. Sadly this was a big resource waste since a lot of time was spent debugging our implementation to find the problem.

In retrospect we realize that choosing the validator of LinkEHR might not have been the best decision since it has proven to contain more faults than we first thought. Archetypes that are not completely correct, according to LinkEHR's archetype model, are sometimes edited without the user's knowledge, for example when at-codes are missing in the ontology. The structure of archetypes is also sometimes changed to pass the validation, for instance a CodePhrase object is always changed into a CComplexObject for validation purposes. The ADL structure is changed in a way that is not according to ADL 1.4. The origin of these problems is essentially that LinkEHR doesn't follow the 1.0.1 specifications because they model archetypes in a different way. To understand how and why LinkEHR model archetypes read (32).

The changes being done to archetypes can be seen by opening an archetype containing at least one CDomainType object (COrdinal, CCodedText and CQuantity) with LinkEHR version 0.8 (42) and then opening the whole archetype tree and selecting "Expand Domain Type" for at least one of the CDomainType objects. When this is done the archetype needs to be saved and then the changes can clearly be seen in the ADL file. Ontology items have been added and for example CodePhrase is changed into a CComplexObject with two attributes, code_string and terminology_id. One example of archetype that can be looked at for verification purposes is openEHR-EHR-ITEM_TREE.oxygen.v1draft.adl (13 p. revision 4906).

The alternative not to go with the LinkEHR's validation procedures did not exist because of time limitations. However one alternative would be to first rigorously test ADL Workbench. If it proves to be better than LinkEHR's validation an implementation in Java should be started. This would not fit into the timeframe for the thesis but it would be beneficial for future study.

### 5.2.1 Changes done to integrate the LinkEHR validator

A number of changes and additions have been done to integrate the validator of LinkEHR into the openEHR Java reference implementation project and the major changes are described in this chapter.

#### 5.2.1.1 ArchetypeConstraint

A new field with the name *parent* was added to the class ArchetypeConstraint and the validator uses this when traversing the archetype structure. The *parent* field keeps track of which ArchetypeConstraint is the parent.

The class CObject have a *parent* field of type CAttribute and the class CAttribute have a list of children of type CObject which can contain zero to * amount of children where * can be a large number. The problem with this structure is that it is impossible to go up in the structure for more than one level. If your current object is of type CAttribute the parent CObject can be seen but if your current object is of type CObject there is no way to know its parent unless you start over from the start of the archetype and go down to its parent with the help of the path field. Doing this each time you want to traverse one step up in the archetype structure is neither effective nor easy; therefore we think a field *parent* in the super class of CObject and CAttribute is a good idea. Of course a method could be created which can get the parent by cutting one level away from the path of the current object and then traverse down to that path of the tree but this solution is not very time effective.

This *parent* field is not only useful for validation purposes but also when editing archetypes. The archetype class contains a method called getNodeAtPath which returns the object that exists at the given path. With this method it's easy to get any object in the structure and from there it facilitates to traverse both down and up in the tree if more modifications are needed.

Two methods called isRecursive was added to the ArchetypeConstraint class because the validator needs these to check if the archetype structure is "recursive". If the archetype is "recursive" the validator is unable to validate it. The archetype is considered "recursive" if the same path exists in more than one place in the archetype structure. The reason quotation marks are used around the word recursive is because no archetype is really recursive, it only means that the path is incorrect since two objects can't have the same path in a tree.

#### 5.2.1.2 Clone

Clone methods have been added to most of the classes that are part of the definition section of the archetype. The clone methods exist for two reasons. First of all, the validation methods testContainment and testContainmentRec uses clone to be able to make some changes to the archetype structure for validation purposes. The validation procedure does not leave the original archetype untouched and therefore the second reason for using clone is to preserve the state of the original archetype during validation.

Something to note at this point is that all clone methods does not make perfect clones of the object they are part of. Most of the clone methods have been copied from the LinkEHR project and they made the clone objects contain only as much information as is necessary for validation. So if the clone methods are used for any other purpose than validation you have to know that it won't be a perfect clone. However, some of the clone methods have been changed into making real clones but since it was not a priority some of the clone methods have been left incomplete.

One of the reasons we wanted to clone the whole archetype before validation is because LinkEHR requires that all the nodeIDs has values and that is not according to AOM 1.0.1 so instead of filling in all the nodeIDs of the original archetype it is instead done to a cloned archetype.

The validation procedure also adds at-codes for an excessive number of nodes and for every one of these at-codes an empty ontology item is added to the ontology. By cloning the whole archetype before sending it to the LinkEHR validator this was avoided.

## 5.3 Development process

This section presents and evaluates the development process.

### 5.3.1 Evaluation of the development process

In this project no concrete requirement contract existed, only some verbal demands from the examiner. This meant that there only was a broad frame of requirements and everything within that was a free choice. The development process in this project can briefly be described by the following list that is more or less chronological. All list-items are described in detail below the bullet list.

- Understanding openEHR specifications, especially the AOM
- Try existing editors for inspiration.
- Decide which way to go, in this case use the Java implementation lead by Rong Chen and extend it to fulfill the needs that were given by the examiner.
- Implement the API iteratively and add features from existing applications.
- During the process stay tuned for new features.
- Perform tests parallel to the implementation phase.
- Write an examination document.
- Try to implement validation.

- Design a GUI proposal.
- Finish implementation of settable immutability.

The first step in this process is intuitive because the project is supposed to follow the specifications that are specified by openEHR. The consecutive step is also important for understanding what has been done in the area and to get inspiration for the API. This step may also lead to a non-creative path because one can assume that the existing work has been carefully thought through, and it can possibly stop a developer from create new solutions. In this project there existed only one real starting point and it was to build on the openEHR Java reference implementation project. The other choice was to create a new Java implementation but the existing implementation already followed AOM version 1.0.1 so it was a very good start. Creating a new Java implementation would also require a huge time frame.

Since the implementation lead by Rong Chen was immutable the first task was to make it more mutable, as that was one of the requirements for the master thesis project. However not to lose the option for immutability it was decided to make immutability settable in the same manner as the Zilics Models project (43).

The fourth bullet describes the implementation and here there were some actions that could be criticized. The fact that many wanted features existed in LinkEHR created some big problems since LinkEHR didn't follow the specifications. Many of these methods were hard to implement because most of the time there wasn't much compatibility between LinkEHR's version and the openEHR Java reference implementation project. Some of the methods required deep understanding of the LinkEHR system which required time. The implementation of validation for example demanded a lot more time and effort than expected. The lesson learned from the implementation phase is to not underestimate the time needed to understand and correctly integrate a solution from one specification to another.

The fifth bullet explains the stay-updated aspect of the working process. To be updated is an essential part of the process especially when the domain is open source and development of new features happens constantly. The impact that new features actually have, changes during the working cycle. For instance the impact of a new feature is huge in the beginning phase of a project because then it's easier to implement and adapt to the changes. Later on in a working cycle it's much harder to implement because then it needs more rescheduling and reprioritizing. Zilics Models presented their new software late in our project. This forced us to ignore some of their solutions but we incorporated the same settable immutability as they have because we think that is a good way to get some benefits of having immutable objects.

An important matter regarding the examination document is to write parallel to the implementation. Many times the implementation phase takes too much focus away from the documentation and that can result in the document being written mostly in the end-phase rather than during the implementation phase when the information is fresh. A good rule is to at least write abstracts in the document whenever topics or interesting information pop up. We managed to work on the report parallel with the development and implementation quite well but we think that it is mainly because we often separated the workload based on a well thought time schedule. There is of course always room for improvements.

The test phase should be executed in parallel with the implementation phase to find errors and problems early. Testing fresh code is important especially when methods and classes depend on each other.

## 5.4 Usefulness

The usefulness of the API is not as high as our own hopes were at the start of the project. The validation procedure is not working correctly and many methods related to improving the way in which parts of the archetype are edited have not been implemented. This is mainly because a lot more resources had to be devoted to trying to fix the validation than we initially thought. More about this can be read in chapter 5.8.

The API is an improvement of the openEHR Java reference implementation project but simply not as improved as we had hoped. If the validation were to be completed there would a clear improvement of the existing Java implementation.

We know where some of the problems lie within the current validator but because of time limitations these cannot be investigated further. This is something that could and should be done given more resources.

## 5.5 Further development

Given the chance to continue working on this project we would first focus on implementing a new semantic validator. This could be done in two ways. The first way is to compare different validation methods, two possible software solutions with semantic validation are LinkEHR and the ADL Workbench. The LinkEHR validation and the ADL Workbench validation have some differences. Some archetypes are valid according to ADL Workbench and the

same archetype might not be valid according to LinkEHR. This is something that has to be investigated more thoroughly.

It is important to have a collection of absolutely correct archetypes and then validate them with the different validation methods. With this information one can at least be able to say that one or more validation methods are better than others and a validation method could then be chosen for implementation. One could assume that the list of archetypes found at the openEHR official website (44) are correct but this is however not the case. One example of an incorrect archetype downloaded from openEHR's official website (45 s. revision 4906) is the archetype openEHR-EHR-OBSERVATION.body_weight-birth.v1 which is supposed to be a specialization of the archetype openEHR-EHR-OBSERVATION.body_weight.v1. Here the weight archetype is defined only for the quantity kg (kilogram) but the weight-birth is defined for both the quantity kg (kilogram) and g (gram), this makes the weight-birth archetype more general than weight. This makes it impossible for the archetype weight-birth to be a specialization of the archetype weight.

The second way to get a new semantic validator is to start from the beginning and design a validator. This could be done by designing a new validation algorithm or follow one of the available algorithms. According to Jose Alberto Maldonado (34) LinkEHR uses a generalized algorithm inspired by Taxonomy of XML Schema Languages Using Formal Language Theory (31). Ideally this should be done without compromising the AOM specification 1.0.1.

There are probably a few useful methods for manipulating archetypes missing in the current API. This mostly has to do with the fact that we were unable to start a real GUI implementation because of time constraints and instead thought about a theoretical GUI which is good but it doesn't give as much as a real GUI would. If a real GUI development would commence we are certain a few more helpful methods would evolve.

## 5.6 Examples of methods needed for a GUI implementation

Here are some examples of actions a user might want to take when creating or editing an archetype. What functionality a user might want and which methods need to be used in different situations are described.

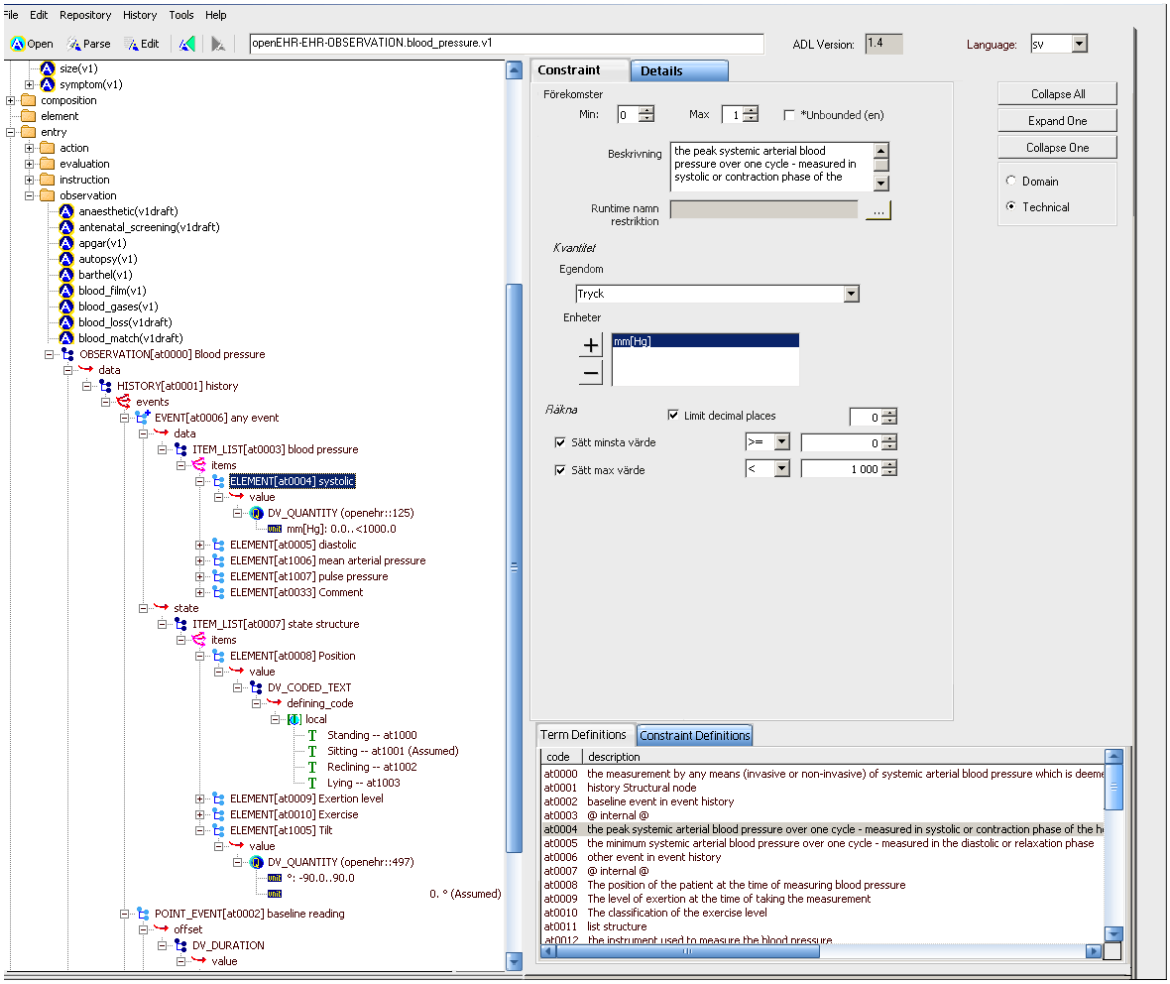


**Figure 14: A simple prototype of how our theoretical GUI looks. (This figure is just a mockup and not an actual GUI)**

### 5.6.1 General thoughts about a GUI implementation

We think that the best approach to design a good GUI would be to separate it completely from the API. This is done to make maintenance easier because it is possible to make changes to the GUI or the API without affecting each other. A design pattern that does this is MVC (Model-View-Controller). The MVC design pattern is described in chapter 2.4.

### 5.6.2 Copy-paste

To be able to copy a CObject from one archetype to another archetype or within the same archetype, all that needs to be done is:

- When right clicking on a CObject in the tree representation a menu is shown and copy is one of the options. Alternatively press Ctrl+C to copy. The GUI can see what object has been pressed by having a mouse listener and send an event when a right click is done.

- The CObject is copied by using the method clone for that object. If this for some reason is not possible the method getObjectAtPath can be run first to get the object and then calling the method clone for that object.
- The GUI saves this cloned object in a field.
- When a location for pasting the object have been chosen either right click and choose paste from the menu or press Ctrl+V.
- When this is done the GUI can see what object is marked and a check have to be done to see if it is legal to paste the object at this location. For instance it is legal if the object being marked is a CAttribute and the CObject will be added to the list of the CAttribute children.
- The field with the cloned CObject does not need to be emptied since the user might want to paste it on many different locations.

### 5.6.3 The archetype tree structure

Similar icons as the ones ADL-workbench uses because ADL-workbench is a popular and widely used product and users are familiar with its symbols and icons.

- The whole archetype needs to be traversed. The method completeNodeId does this but not to the depth required for the whole tree to be drawn although the changes needed for that is minimal.
- A mouse and a keyboard listener need to be connected to the archetype tree to be able to register what object the user wants to manipulate.
- For every different kind of object a list of possible actions need to be listed when right clicking on the object.

### 5.6.4 Open

To be able to open an archetype everything that needs to be done is with the GUI select a path to the file and then pass this as a string to the method loadADL from the package org.upv.ibime.linkehr.io from the class IO.java. Examples of this can be found in the eclipse project named Archetype API test.

### 5.6.5 Save

To be able to save the serialize class needs to work properly, this has not yet been tested fully.

### 5.6.6 Edit

A common thing to notice in most available archetype editors is that they hide parts of the Reference Model (RM). This is not especially pedagogical since the specifications contain a lot that is left unused. We feel that it is important

to give the users of archetype editors more freedom in editing more settings that are available in the RM than most current editors offer. Examples of these settings are participation and attestation that are described in the section Root below.

When clicking on an object in the tree all fields that are supposed to be editable are shown in another window, located to the right in Figure 14. For fields that have only a few specific values a drop down menu should be available. This helps the user by providing information the user might not have by showing all possible options and also prevents the user from making a wrong entry.

#### 5.6.6.1 Common for most objects

A description of the object corresponding to the ontology should exist as well as bindings to terminologies. These areas should also be editable.

#### 5.6.6.2 Root

When the root node of the archetype is selected general options like setting participation should be available, i.e. what participant is active in a certain action and in what way the result is recorded. Attestation should also be able to be set to force a particular healthcare agent to sign for a particular event, i.e. to force the doctor sign for authorization to give the patient a particular item on the list of medicine. For a detailed list of what data should be stored for participation and attestation read the Common IM document (46).

#### 5.6.6.3 CComplexObject

Specify the number of occurrences of this object that can exist in the archetype. We think that it is a good idea to show more than just the settings for the CComplexObject when it is selected. Settings for the CAttribute or the CAttributes should be shown even when the CComplexObject is selected. For example when the CComplexObject systolic is selected in an archetype blood pressure is selected, then we think it is a good idea if settings for the CSingleAttribute DvQuantity is shown, i.e. that the unit of pressure can be set and what values are allowed.

#### 5.6.6.4 CSingleAttribute and CMultipleAttribute

Option to make the attribute mandatory or optional. If a new attribute is added to a single attribute it is changed into a CMultipleAttribute.

For CMultipleAttribute cardinality settings shall be available, setting if members are supposed to be ordered, if they are unique and also setting the number of attributes.

## 5.7 Delimitations

One limitation of our API is that we don’t have any methods related to templates implemented; this forces the user to implement these themselves or to create an application without template support. Another limitation is that the API is written in Java so all GUIs created using this API has to be written in a Java compatible environment.

The goal was to implement a complete API with a very simple GUI but since this project is restricted to a master thesis time frame some delimitations have been done. Firstly the API is only managing archetypes and there are no template features. Secondly the GUI part is delimited to a more theoretical level where only design principles are presented.

## 5.8 API limitations

At the moment the API can only parse and serialize ADL files. There are classes for making this possible for XML files as well but this has not yet been fully tested. The API does not have template functionality which is something that will have to be implemented in the future to have a more complete EHR system.

The validation functionality in the API is not yet fully functional and a reason for this is because LinkEHR have not released their source code for their working validation procedure that can be seen in their runnable binary file, and since this was not documented anywhere time was spent trying to correct the validation procedure. Right now (2009-01-13) the validation procedure is not yet fully functional but the ability to run validation is complete. What needs to be done is to implement the validation methods that can be observed in the LinkEHR editor, for example the “Expand domain type” method that is needed to validate an archetype in the LinkEHR implementation.

## 5.9 Why LinkEHR?

The reason a lot of code from the project LinkEHR was used is because this was the only Java implementation with a seemingly good semantic validation and also because we liked some of their design decisions, for example to use a *parent* field that connects all archetype constraint objects which makes traversing the object structure easier which is important for a mutable model. Of course some other semantic validation could be used that was not yet

implemented in Java and a good choice for this would be the ADL workbench. The reason we did not choose to solve the validation this way was because our task was to develop an API for creating and editing archetypes and not specifically focus on validation. Even though we realized the importance of validation there was not enough time available to port the validation procedures of ADL workbench from the Eiffel programming language to Java.

By implementing the validation code a lot more changes followed to many of the AM classes. The major of these changes are described in chapter 5.2.1.

## 5.10 A restart with gained knowledge

Given the chance to go back in time and start over but not lose the knowledge we have gained during this time we would do some things different. Here is a description of some of the things we would do.

We think that the best approach to get helpful API methods is to start from the GUI side by making small prototypes or just simply draw a GUI manually. When doing this you need to think about what needs to be done for every possible user action that one might want. For every action there are a number of sub actions, how many they are depends of course on how advanced the user action is. In some cases the whole user action needs to be a separate method but most of the time it is very good to also make most of the sub actions into separate methods because many of them will probably be used for other user actions.

For the validation we would not implement the LinkEHR validator. There are three reasons for this. Their validator is very hard to understand so if this validation process is to be used a good start would be to redesign the code by following the same validation algorithms they use which is a generalized algorithm inspired by Taxonomy of XML Schema Languages Using Formal Language Theory (31) according to Jose Alberto Maldonado (34).

The second reason we rather not use their validation method is because they validate against RM objects created as separate ADL files. We think this creates too much maintenance work, when implementing their validator we were for example forced to edit these files to be able to get them to pass through the ADL parser. This was because the parser was now in a later version and was not as accepting as older versions. This would be a problem that would come up every time additions are made to the ADL parser because then corrections would have to be made in the RM ADL files.

The third reason is because LinkEHR deviates too much from AOM 1.0.1 and that has created some problems when integrating their validation procedures into our API.

Instead we think it is a good idea to investigate how well the validation in ADL Workbench is working and if it proves to be a good validation method it should be ported from Eiffel to Java. This could of course be a big task and might even be a thesis on its own. It would also be good if some elements of the RM validator from the openEHR Java reference implementation project were incorporated into it. The warning and error handling for example are very good and helpful. Also some elements of the ZM (Zilics Models) validation might be incorporated. This is something that needs more investigation since we have not been able to check their implementation in any depth because of the time frame of this thesis.

# 6 Conclusion

The developed API can create and edit archetypes and it contains more functionality than the kernel it is based on (the openEHR java reference implementation project). The API is licensed under open source, is platform independent and follows the openEHR specifications with the exception of some documented deviations, which means that continued development is possible even if the openEHR specifications needs to be followed to the letter. The deviations from the specifications are there because of the validation functionality and they are only additions and all previous functionality still exists. This means that the API would be compatible with any software following AOM 1.0.1.

The openEHR concept is very complex and a lot of time has to be devoted to studying the specifications. To help users understand the API it has been well documented and follows code standards. Examples have also been included in a separate Eclipse project to further help the continuation of this project.

Primary goals of the master thesis project where achieved but one of them are worth mentioning. The primary goal for the API to include some validation functionality have been achieved, however the validation is not working as well as we hoped. The API is not reliable for complete semantic validation for archetypes at this point. We suggest that the validation procedures of ADL workbench is evaluated for the possibility of translating some of its validation functionality into Java code for continued development. Read more about this in chapter 5.8 API limitations and chapter 5.10 A restart with gained knowledge.

A requirement was to have mutable objects but instead of removing the immutability functionality the ability to set objects to an immutable state exists.

The secondary goals of this project, to implement template functionality and develop a simple GUI, have not been fully completed because of time constraints. Template functionality was exchanged for improved edit functionality and to give more time to the integration of validation functionality. This decision was made because all included parties of the project thought it was more important to have a solid API with the ability to create and edit archetypes before the implementation of template functionality is started. The other secondary goal, to implement a simple GUI, was delimited to give a GUI proposal which can be found in chapter 5.6 Examples of methods needed for a GUI implementation.

This master thesis project have been fun and very educational and we are happy to have been a part of the development of this area.

# 7 References


1. **openEHR Foundation.** openEHR. *openehr.org.* [Online] openEHR Foundation. http://www.openehr.org/home.html.

2. **Beale, Thomas.** Archetype Object Model. *OpenEHR.* [Online] 2007. [Cited: 10 11, 2008.] http://www.openehr.org/releases/1.0.1/architecture/am/aom.pdf.

3. **Y, T L.** Information modeling from design to implementation, National Institute of Standards and Technology. [Online] 1999. [Cited: 10 5, 2008.] http://www.mel.nist.gov/msidlibrary/doc/tina99im.pdf.

4. **(ISO), International Organization for Standardization.** About ISO. *International Organization for Standardization.* [Online] [Cited: 11 17, 2008.] http://www.iso.org/iso/about.htm.

5. **Sierra, Kathy and Bates, Bert.** *Head first java.* Gravenstein Highway Nortyh, Sebastopol : O'Reilly Media, 2005.

6. **Nielson, Hanne and Nielson, Riis.** *Semantics with Applications , A Formal Introduction.* Chicester, England : John Wiley & Sons, 1995.

7. **Wells, Don.** Extreme Programming. *Extreme Programming.* [Online] 02 17, 2006. [Cited: 10 03, 2008.] http://extremeprogramming.org/.

8. **CollabNet, Inc.** Subversion. *Tigris.org.* [Online] den 13 september 2008. http://subversion.tigris.org/.

9. **CollabNet.** Subclipse. *Tigris.org.* [Online] [Cited: 10 10, 2008.] http://subclipse.tigris.org/.

10. **CollabNet, Inc.** TortoiseSVN. *Tigris.org.* [Online] [Cited: 09 13, 2008.] http://tortoisesvn.tigris.org/.

11. **Sun Microsystems.** Java Code Conventions. *sun.com.* [Online] 10 12, 1997. [Cited: 09 10, 2008.] http://java.sun.com/docs/codeconv/CodeConventions.pdf.

12. **openEHR Foundation.** Archetype Definitions and Principles. *openEHR.org.* [Online] 03 17, 2007. [Cited: 09 03, 2008.] http://www.openehr.org/releases/1.0.1/architecture/am/archetype_principles.pdf.

13. —. Archetypes. *openEHR.* [Online] 09 05, 2008. [Cited: 10 13, 2008.] http://www.openehr.org/wsvn/knowledge/archetypes/dev/adl/openehr/ehr/.

14. *Archetypes: Constraint-based domain models for future-proof information systems.* **Beale, Thomas.** Washington, Seattle, USA : Northeastern University, 2002.

15. **openEHR Foundation.** Archetype Definition Language. *openehr.org.* [Online] 03 13, 2007. [Cited: 09 13, 2008.] http://www.openehr.org/releases/1.0.1/architecture/am/adl.pdf.

16. —. The Template Object Model (TOM). *openehr.org.* [Online] 03 13, 2007. [Cited: 09 15, 2008.] http://www.openehr.org/releases/1.0.1/architecture/am/tom.pdf.

17. **ComputerWorld.** Quickstudy: Application Programming Interface (API). [Online] Computerworld Inc. [Citat: den 20 11 2008.] http://www.computerworld.com/action/article.do?command=viewArticleBasic&articleId=43487.

18. *How to Design a Good API and Why it Matters.* **Bloch, Joshua.** San Diego, California : OOPSLA, 2005.

19. **Sun Microsystems Inc.** Java Blueprints Model-View-Controller. *sun.com.* [Online] Sun Microsystems, 2002. [Cited: 11 05, 2008.] http://java.sun.com/blueprints/patterns/MVC-detailed.html.

20. **Mozilla.** Mozilla Public License Version 1.1. *Mozilla.* [Online] Mozilla Foundation. http://www.mozilla.org/MPL/MPL-1.1.html.

21. **GNU operating Systems.** The GNU General Public License. [Online] Free Software Foundation (FSF), den 18 01 2009. [Citat: den 07 02 2009.] http://www.gnu.org/licenses/gpl.html.

22. **GNU Operating Systems.** GNU Lesser General Public License. [Online] Free Software Foundation (FSF), den 28 01 2009. [Citat: den 07 02 2009.] http://www.gnu.org/copyleft/lesser.html.

23. **Chen, Rong.** openEHR Reference Model Java ITS. *openehr.org.* [Online] 09 02, 2006. [Cited: 10 03, 2008.] http://www.openehr.org/releases/1.0.1/its/Java/openEHR-JavaITS.pdf.

24. **Sun Mircrosystems Inc.** How to Write Doc Comments for the Javadoc Tool. *sun.com.* [Online] 2004. [Cited: 10 12, 2008.] http://java.sun.com/j2se/javadoc/writingdoccomments/.

25. **Institute of Electrical and Electronics Engineers, Inc.** SOFTWARE QUALITY. *swebok.* [Online] 2007. [Cited: 11 10, 2008.] http://www.swebok.org/ch11.html#Ref2.1.

26. **Food and Drug Administration (FDA).** General Principles of Software Validation. *FDA.* [Online] 06 09, 1997. [Cited: 10 15, 2008.] http://www.fda.gov/cdrh/comp/guidance/938.html#_Toc517237938.

27. **Eushiuan, Tran.** Verification/Validation/Certification. [Online] 1999. [Cited: 10 10, 2008.] http://www.ece.cmu.edu/~koopman/des_s99/verification/index.html.

28. **Müller, Thomas, et al.** Certified Tester. *ISTQB.* [Online] 2007. [Cited: 11 11, 2008.] http://www.istqb.org/downloads/syllabi/SyllabusFoundation.pdf.

29. **Gottfried, Paul.** What Is Software Failure? *IEEE Transactions on Reliability.* September, 1996, Vol. 45, 3.

30. **LinkEHR.** openlinkehr.zip. *pangea.upv.es.* [Online] [Cited: 08 20, 2008.] http://pangea.upv.es/linkehr/wp-content/uploads/2008/05/openlinkehr.zip.

31. **Makoto, Murata, et al.** Taxonomy of XML schema languages using formal language theory. *acm.org.* [Online] 11 2005. [Cited: 11 05, 2008.] http://portal.acm.org/citation.cfm?id=1111631&coll=GUIDE&dl=GUIDE&CFID=11954650&CFTOKEN=87461729&ret=1#Fulltext.

32. **Maldonado, Segura, et al.** *Framework for clinical data standarization based on archetypes.* s.l. : World Congress on Health (Medical) Informatics (Medinfo'07), 2007. 978-1-58603-774-1.

33. **Kuper, Gabriel M and Siméon, Jérôme.** Subsumption for XML Types. [Online] [Cited: 11 15, 2008.] http://xml.coverpages.org/kuper-subsumption.pdf.

34. **Maldonado, Jose Alberto.** e-mail. 2008.

35. **Goetz, Brian.** *Java theory and practice: To mutate or not to mutate?* [Online] IBM. [Citat: den 10 12 2008.] http://www.ibm.com/developerworks/java/library/j-jtp02183.html.

36. **Eiffel.** Eiffel in a Nutshell. *Eiffel Software.* [Online] Eiffel Software. [Citat: den 14 01 2009.] http://archive.eiffel.com/eiffel/nutshell.html.

37. **Minella, Michael.** Unit testing with JUnit and EasyMock. *michaelminella.com.* [Online] [Cited: 11 10, 2008.] http://www.michaelminella.com/testing/unit-testing-with-junit-and-easymock.html.

38. **Sun Microsystems.** JUnit Reloaded. *java.net.* [Online] Sun Microsystems, 12 07, 2006. [Cited: 01 14, 2009.] http://today.java.net/pub/a/today/2006/12/07/junit-reloaded.html.

39. **Chen, Rong.** e-mail. 2008.

40. **LinkEHR.** LinkEHR Project. *LinkEHR.* [Online] http://linkehr.net/.

41. **Klasson, Filip and Väyrynen, Patrik.** SVN server. [Online] https://svn.imt.liu.se/mi/openehr/Archetype API - master thesis/.

42. **LinkEHR.** LinkEHR version 0.8. [Online] [Citat: den 20 09 2008.] http://pangea.upv.es/linkehr/wp-content/uploads/2008/10/linkehr-edv08cenversion.zip.

43. **Zilics models.** Zilics models. [Online] http://www.openehr.org/svn/ref_impl_java/SANDBOX/zilics-models/.

44. **OpenEHR.** openEHR Archetypes. *openEHR.* [Online] [Citat: den 10 12 2008.] http://www.openehr.org/clinicalmodels/archetypes.html.

45. —. observation. [Online] OpenEHR foundation. http://www.openehr.org/svn/knowledge/archetypes/dev/adl/openehr/ehr/entry/observation/.

46. **openEHR Foundation.** Common Information Model. [Online] 04 08, 2007. [Cited: 11 11, 2008.] http://www.openehr.org/releases/1.0.1/architecture/rm/common_im.pdf.

# 8 Appendix

## 8.1 Implemented methods

This section presents all new methods in the archetype class, i.e. the classes that are added to the original archetype class in the openEHR Java reference implementation project. In the description of each method only the direct help functions that are used are mentioned. There are also methods that are only introduced in the code but not implemented. These methods are created because they exist in AOM 1.0.1. Many other changes and additions were made to the kernel but these were related to making the API mutable. These changes are mostly about making certain constructors and fields public as well as creating set methods. Other changes was to implement updates for different kinds of maps like nodemap, pathmap and hash-maps.

### 8.1.1 The Archetype class

```
public Archetype(String id, String parentId, String
conceptCode, TerminologyService terminologyService)
This method constructs an archetype with using the
full constructor with some default values.

public boolean removeCObjectAtPath (String path)
This method removes the CObject at the given path
and removes all the unique at/ac-codes in the
removed “branch” from the archetype ontology. This
method works only on CObjects with parents so it is
impossible to remove the root node with this method,
in that case the archetype definition can be set to
null. The method uses many help-functions including
getCodesToRemove, getNodeByPath, deleteTerm etc. The
method returns a boolean if the removal succeeded or
not.

public void correctParentsInArchetype()
This method traverses the archetype definition and
sets parent to all objects that are instances of
ArchetypeConstraints (CObjects and CAttributes).
This method is mainly created to set parents on
archetypes that are generated from markup languages,
```

like ADL or XML (input via XML is currently not implemented in the API).

**public boolean** isValidArchetype()
This method returns a boolean whether the archetype is valid or not. This method checks if the archetypes constraints are narrower or identical to its RM type. The method creates a clone of the archetype and manipulates the clone because the help methods for the validation by LinkEHR will alter the structure. This method uses help functions like correctParentsInArchetype, completeNodeID, refreshPath and testContainment (in the class validator).

**public void** setArchetypeId(ArchetypeID archetypeId)
This method sets the archetypeID.

**public void** setConceptCode(String conceptCode)
This method sets the concept attribute to the given conceptCode.

**public** String getOntologyNodeText(String code)
This method returns the node text from ontology given its identifier.

**public** String getNodeText(String lang, String code)
This method returns the node text from ontology given its identifier and language.

**private void** traverseTree(ArchetypeConstraint node)
A variation of this method is used in all methods concerning traversing the definition part of the archetype. The main method traverseTree simply wanders through every node in the given ArchetypeConstraint and sets the parent field of every child.

**private** List<String>
traverseTreeCollectCodes(ArchetypeConstraint node)
This method returns all at/ac-codes in the given object node as a list of strings. Uses traverseTreeCollectCodeshelp as a help function.

**private void**
traverseTreeCollectCodeshelp(ArchetypeConstraint

node, List<String> codelist)
This method traverses the definition tree adds at/ac-codes to a the given codelist.

**private** List<String> getCodesToRemove(String path)
getCodesToRemove returns a list of strings with the at/ac-codes that are unique for the branch which root node has the given path. The method uses the paths of the nodes to identify the at/ac-codes.

**private** List<String> getCodesToRemove(String path)
This method returns all unique at/ac-codes that exists in the branch whose root is the object at given path. This method uses help-functions like traverseseTreeCollectCodes and getParent.

**public void** completeNodeID()
This method fills the node ID attribute of all CObjects and also sets new paths for these objects. The help method that is used is getNextAvailiableTermCode.

**private int** archetypeDepth()
This method returns the specialization depth of the archetype. This is a help function to searchForUsedAtCodes.

**private** String getNextAvailableTermCode()
This method returns the next available at-code. It uses searchForUsedAtCodes as help method.

**private** LinkedList<Integer> searchForUsedAtCodes()
This method that returns all at-codes in the definition as a LinkedList.

### 8.1.2 The ArchetypeOntology class

**private void** deleteConstraintBinding(String code)
This method deletes all constraint bindings in the ontology that refers to the given ac-code.

**private void** deleteConstraintDefinitions(String code)
This method deletes all constraint definitions in

the ontology on all languages with the given ac-code.

**private void** deleteTermBinding(String code)
This method deletes all term bindings in the ontology that refers to the given at-code.

**private void** deleteTermDefinitions(String code)
This method deletes all term definitions in the ontology on all languages with the given at-code.

**public void** deleteTerm (String code)
Simple help method that simply calls the two methods private methods deleteTermBinding and deleteTermDefinitions and by doing that deletes the term.

**public void** deleteConstraint (String code)
Simple help method that simply calls the two methods private methods deleteConstraintBinding and deleteConstraintDefinitions and by doing that deletes the constraint.

## 8.2 JUnit example

These junit tests are written to check the method removeCObjectAtPath.

```
@Test
public void testremoveCObjectAtPathNodeRemoval() {
ta.correctParentsInArchetype();
// Test remove Archetype_Slot
ta.removeCObjectAtPath("/activities[at0001]/description");
assertEquals(((CComplexObject)
ta.getNodeAtPath("/activities[at0001]")).getAttributes().get(0).getChildr
en().isEmpty(), true);
// Test remove CPrimitiveObject
ta.removeCObjectAtPath("/activities[at0001]/action_archetype_id");
assertEquals(((CComplexObject)
ta.getNodeAtPath("/activities[at0001]")).getAttributes().get(1).getChildr
en().isEmpty(), true);
// Test remove Archetype_Internal_Ref
ta.removeCObjectAtPath("/activities[at0001]/data");
assertEquals(((CComplexObject)
ta.getNodeAtPath("/activities[at0001]")).getAttributes().get(2).getChildr
en().isEmpty(), true);
// Test remove Constraint_Ref
ta.removeCObjectAtPath("/activities[at0001]/reference");
```

```
assertEquals(((CComplexObject)
ta.getNodeAtPath("/activities[at0001]")).getAttributes().get(3).getChildr
en().isEmpty(), true);
// Test remove CComplexObject
ta.removeCObjectAtPath("/activities[at0001]");
assertEquals(ta.getDefinition().getAttributes().get(0).getChildren().isEm
pty(), true);
}

@Test
public void testremoveCObjectAtPathCodeRemoval() throws
ParseADLException, IOException {
// Test if the method is removing atxxxx-codes from ontology
Archetype arch1 = IO.loadADL("C:\\Exjobb\\Arketyper\\openEHR-EHR-
OBSERVATION.body_mass_index.v1.adl");
arch1.correctParentsInArchetype();
assertNotSame(arch1.getNodeAtPath("/protocol[at0005]"), null);
assertEquals(9,arch1.getOntology().getTermDefinitionsList().get(0).getDef
initions().size());
// Remove branch with codes: at0006/0007/0008, where 7 and 8 are
  within an C_Code_Phrase text
arch1.removeCObjectAtPath("/protocol[at0005]/items[at0006]");
assertEquals(6,arch1.getOntology().getTermDefinitionsList().get(0).getDef
initions().size());
// Remove branch with one code at0005
arch1.removeCObjectAtPath("/protocol[at0005]");
assertEquals(true,arch1.getDefinition().getAttributes().get(1).getChildre
n().isEmpty());
assertEquals(5,arch1.getOntology().getTermDefinitionsList().get(0).getDef
initions().size());
// Remove branch with codes: at0003/0004
arch1.removeCObjectAtPath("/data[at0001]/events[at0002]/data[at0003]");
assertEquals(3,arch1.getOntology().getTermDefinitionsList().get(0).getDef
initions().size());
}

@Test(expected=NullPointerException.class)
public void testNullPointerExceptionremoveCObjectAtPath() throws
ParseADLException, IOException {Archetype arch1 =
IO.loadADL("C:\\Exjobb\\Arketyper\\openEHR-EHR-
OBSERVATION.body_mass_index.v1.adl");
arch1.correctParentsInArchetype();
arch1.removeCObjectAtPath("/protocol/items[at0006]");
((CComplexObject)arch1.getDefinition().getAttributes().get(1).getChildren
().get(0)).getAttributes().get(0).getChildren().get(0);
}

@Test(expected=NullPointerException.class)
public void testNullPointerExceptionremoveCObjectAtPathFalseArgument()
throws ParseADLException, IOException {
Archetype arch1 =IO.loadADL("C:\\Exjobb\\Arketyper\\openEHR-EHR-
OBSERVATION.body_mass_index.v1.adl");
arch1.correctParentsInArchetype();
arch1.removeCObjectAtPath("/protocol/items[at006]/value/");
}
```

## 8.3 LinkEHR validation algorithm

Input: Two schemas (archetypes): $S_1$ and $S_2$.

Output: the set of subsumption functions from $S_2$ to $S_1$.

(1) Let S be an empty stack of lists of sets of type names;

(2) Let PR an empty stack of sets of rules;

(3) Let P be an empty stack of sets of type names;

(4) Let F a set of pairs of type names; //it contains the subsumption functions, $if\ \left(t,t'\right)\in F, then\ \theta\left(t\right)=t'$

(5) Push the root type of $S_1$ into P;

(6) Let $G_1$ = CG($S_1$); CG($S_1$)is the containment graph of the schema $S_1$

(7) Let $G_2$ = CG($S_2$);

(8) Traverse $G_2$ in the depth-first manner

(9) When a node n is visited

(10) Find rules $t_i \coloneqq p_i\{r_i\}$ if n is unordered, $t_i \coloneqq p_i\{r_i\}$ or $t_i \coloneqq p_i\langle r_i\rangle$ otherwise, such that $t_i$ is at the top of P and $label_{S_2}(n)\subseteq p_i$;

(11) IF no such rule can be found THEN report “no function” and halt;

(12) Push the set of found rules to PR;

(13) Push an empty list to S;

(14) Push the set of type names occurring in some $r_i$ into P;

(15) When a node n is existed from

(16) Pop the set of rules $\{t_i \coloneqq p_i\langle r_i\rangle \mid i=1,..,n\}\cup\{t_i \coloneqq p_i\{r_i\} \mid i=n+1,..,m\}$ out of RP;

(17) Pop a list of set of type names $\langle T_1,...,T_k\rangle$ out of S;

(18) Let C = $\{t_1',...,t_k' \mid t_i' \in T_i, 1\le i\le k\}$;

(19) Let T = $\{t \mid (n,t)\in F\}$;

(20) IF T is empty THEN

(21) Let T be the set of $t_i$ such that a list of type names $c \in C$ exists that $\theta_c(Lang(n)) \subseteq Lang(t_i)$;

(22) IF T is empty THEN report “no function” and halt;

(23) Add $(n, t_i)$ to F for each $t_i \in T$;

(24) Append T to the list of sets of type names at the top of S;

(25) Pop a type name set out of P;

(26) Delete all $(t, t')$ from F such that do not exist a path $t_0, t_1, ..., t_n$ in $G_1$ where $t_0$ is the root of $G_1$ and $t_n = t$ and a path $t_0', t_1', ..., t_n'$ in $G_2$ where $t_0'$ is the root of $G_2$ and $t_n' = t'$ which satisfy $(t_i, t_i') \in$ F, $i = 0...n-1$.

(27) Return F;

# 8.4 UML diagram of parts of the original Java implementation

## 8.4.1 ConstraintModel package

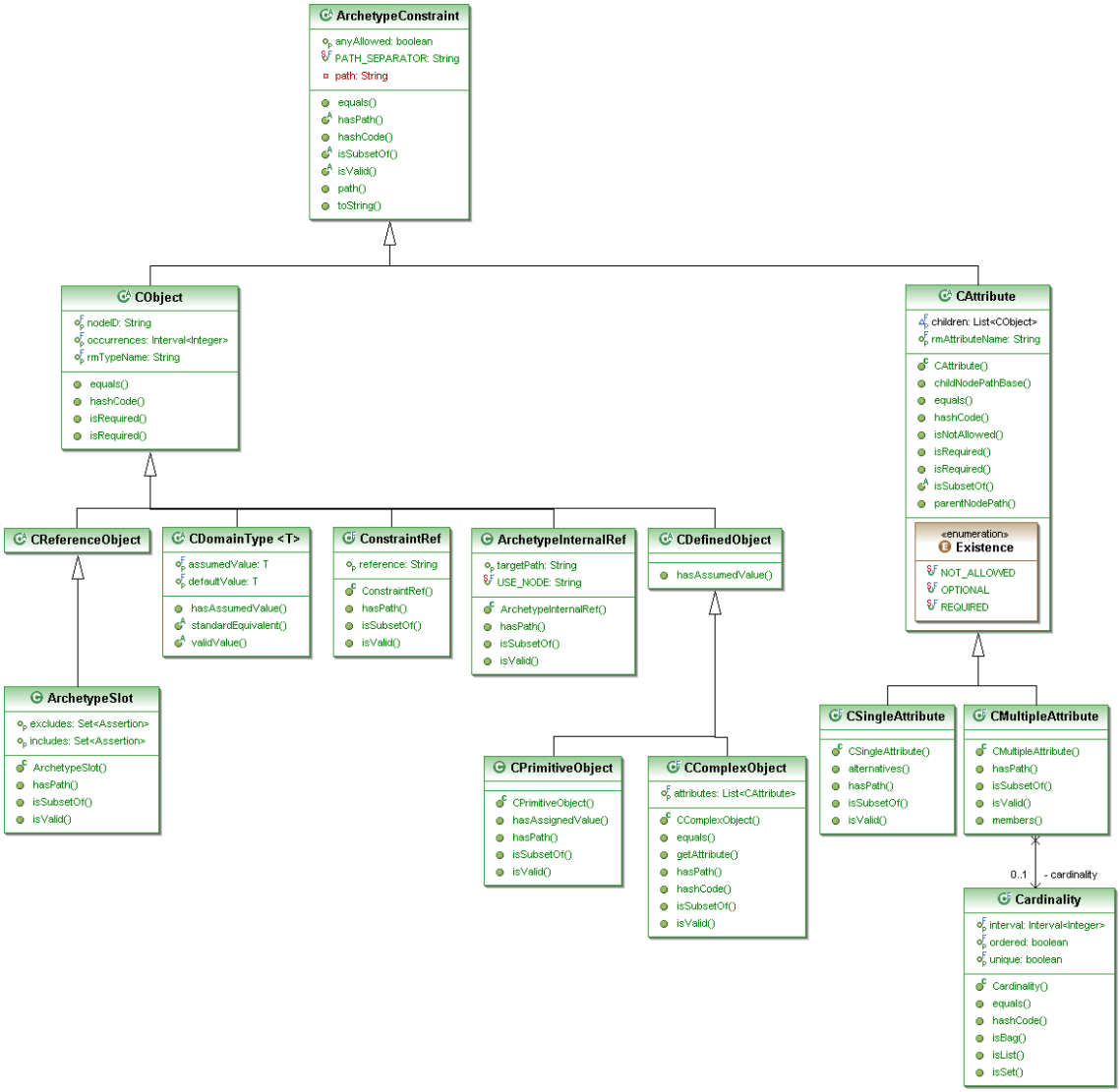


**Figure 15: This is an UML diagram of the constraint model package in the openEHR Java reference implementation project (openehr-aom: revision 430, 2008-09-24). By comparing this diagram with the diagram in Figure 4 you can see that this is not according to AOM 1.0.1.**