Object Data Instance Notation (ODIN)

Editors: {T Beale}[a]		
Revision: 1.5.1	*Pages:* 49	*Date of issue:* 15 Apr 2013
Status: TRIAL		

a. Ocean Informatics

Keywords: object, serialisation, syntax

© 2013- The *open*EHR Foundation

The *open*EHR Foundation is an independent, non-profit community, facilitating the sharing of health records by consumers and clinicians via open-source, standards-based implementations.

Affiliates Australia, Brazil, Japan, New Zealand, Portugal, Sweden

Licence Creative Commons Attribution-NoDerivs 3.0 Unported.
creativecommons.org/licenses/by-nd/3.0/

Support **Issue tracker**: www.openehr.org/issues/browse/SPECPR
Web: www.openEHR.org

Amendment Record

Issue	Details	Raiser	Completed
RELEASE 1.1 candidate			
1.5.1	Separate from ADL spec and rename to ODIN.	T Beale	15 Apr 2013
1.5.0	**SPEC-xxx**. Add fractional seconds to dADL grammar.	T Beale	15 Aug 2012
RELEASE 1.0.2			
1.4.1	**SPEC-268**: Correct missing parentheses in dADL type identifiers. dADL grammar rules updated.	R Chen	12 Dec 2008
RELEASE 1.0.1			
1.4.0	**CR-000213**: Include 'T' in dADL Date/time, Time and Duration values. **CR-000216**: Allow mixture of W, D etc in ISO8601 Duration (deviation from standard).	T Beale S Heard	13 Mar 2007
RELEASE 1.0			
1.3.0	**CR-000143**. Add partial date/time values to dADL syntax. **CR-000149**. Add URIs to dADL and remove query() syntax.	S Heard T Beale	18 Jun 2005
RELEASE 0.95			
1.2	**CR-000110**. Added explanatory material; added domain type support; rewrote of most dADL sections.	T Beale	15 Nov 2004
RELEASE 0.9			
0.9.9	**CR-000075**. Added dADL object model.	T Beale	28 Dec 2003
0.9.8	**CR-000070**. Copyright Assigned by Ocean Informatics P/L Australia to The *open*EHR Foundation.	T Beale, S Heard	29 Nov 2003
0.9.7	Added simple value list continuation (",..."). Changed path syntax so that trailing '/' required for object paths. Remove ranges with excluded limits. Added terms and term lists to dADL leaf types.	T Beale	01 Nov 2003
0.9.6	Additions during HL7 WGM Memphis Sept 2003	T Beale	09 Sep 2003
0.9.5	Renamed dDL to dADL. Changed path syntax to conform (nearly) to Xpath.	T Beale	03 Sep 2003
0.9	Rewritten with sections on dDL.	T Beale	28 July 2003
0.8	Initial Writing	T Beale	10 July 2003

© 2013- The openEHR Foundation
Issue tracker: www.openehr.org/issues/browse/SPECPR Web: www.openEHR.org

Trademarks

“Microsoft” and “.Net” are registered trademarks of the Microsoft Corporation.

“Java” is a registered trademark of Sun Microsystems.

“Linux” is a registered trademark of Linus Torvalds.

Acknowledgements

The work reported in this document was funded by:

- Ocean Informatics;
- Centre for Health Informatics and Multi-professional Education (CHIME), University College London;
- Thomas Beale.

© 2013- The openEHR Foundation
Issue tracker: www.openehr.org/issues/browse/SPECPR Web: www.openEHR.org

© 2013- The openEHR Foundation
Issue tracker: www.openehr.org/issues/browse/SPECPR Web: www.openEHR.org

Table of Contents

© 2013- The openEHR Foundation
Issue tracker: www.openehr.org/issues/browse/SPECPR Web: www.openEHR.org

© 2013- The openEHR Foundation
Issue tracker: www.openehr.org/issues/browse/SPECPR Web: www.openEHR.org

1 Introduction

1.1 Purpose

This document describes the syntax of the Object Data Instance Notation (ODIN), previously known as the 'dADL' syntax from the *open*EHR ADL specifications. It is intended for software developers, technically-oriented domain specialists and subject matter experts (SMEs). ODIN is designed as a human-readable and computer-processible data representation syntax, and can be hand-edited using a normal text editor.

1.2 Nomenclature

In this document, the term 'attribute' denotes any stored property of a type defined in an object model, including primitive attributes and any kind of relationship such as an association or aggregation. Where XML is mentioned, XML 'attributes' are always referred to explicitly as 'XML attributes'.

1.3 Status

This document is under development, and is published as a proposal for input to standards processes and implementation works.

This document is available at http://www.openehr.org/specification/Releases/trunk/architecture/syntaxes/odin.pdf.

1.4 Peer review

Known omissions or questions are indicated in the text with a "to be determined" paragraph, as follows:

TBD_1: (example To Be Determined paragraph)

Areas where more analysis or explanation is required are indicated with "to be continued" paragraphs like the following:

To Be Continued: more work required

Reviewers are encouraged to comment on and/or advise on these paragraphs as well as the main content. Please send requests for information to info@openEHR.org. Feedback should preferably be provided on the mailing list openehr-technical@openehr.org, or by private email.

1.5 Conformance

To Be Continued:

1.6 Tools

To Be Continued:

© 2013- The openEHR Foundation
Issue tracker: www.openehr.org/issues/browse/SPECPR Web: www.openEHR.org

© 2013- The openEHR Foundation
Issue tracker: www.openehr.org/issues/browse/SPECPR Web: www.openEHR.org

2 Overview

The ODIN syntax provides a formal means of expressing *instance data* based on an underlying object-oriented or relational information model, which is readable both by humans and machines. The general appearance is exemplified by the following:

```
person = (List<PERSON>) <
    [01234] = <
        name = <                                    -- person's name
            forenames =     <"Sherlock">
            family_name =   <"Holmes">
            salutation =    <"Mr">
        >
        address = <                                 -- person's address
            habitation_number = <"221B">
            street_name =   <"Baker St">
            city =          <"London">
            country =       <"England">
        >
    >
    [01235] = <
        -- etc
    >
>
```

In the above the attribute names `person`, `name`, `address` etc, and the type `List<PERSON>` are all assumed to come from an information model. The `[01234]` and `[01235]` tags identify container items.

The basic design principle of ODIN is to be able to represent data in a way that is both machine-processible and human readable, while making the fewest assumptions possible about the information model to which the data conforms. To this end, type names are optional; often, only attribute names and values are explicitly shown. No syntactical assumptions are made about whether the underlying model is relational, object-oriented or what it actually looks like. More than one information model can be compatible with the same ODIN-expressed data. The UML semantics of composition/aggregation and association are expressible, as are shared objects. Literal leaf values are only of 'standard' widely recognised types, i.e. Integer, Real, Boolean, String, Character and a range of Date/time types. In standard ODIN, all complex types are expressed structurally.

The ODIN syntax as described above has a number of useful characteristics that enable the extensive use of paths to navigate it, and express references. These include:

- each <> block corresponds to an object (i.e. an instance of some type in an information model);
- the name before an '=' is always an attribute name or else a container element key, which attaches to the attribute of the enclosing block;
- the formal type of leaf values can be inferred purely from syntax;
- paths can be formed by navigating down a tree branch and concatenating attribute name, container keys (where they are encountered) and '/' characters;
- every node is reachable by a path;
- dynamically bound types can be explicitly indicated;
- shared objects can be referred to by path references.

© 2013- The openEHR Foundation
Issue tracker: www.openehr.org/issues/browse/SPECPR Web: www.openEHR.org

3 Basics

3.1 File Encoding

ODIN files are intended to be capable of accommodating characters from any language, and may be used for multiple languages at once, e.g. as translation files for software. Accordingly, the assumed encoding for an ODIN file is UTF-8 unicode.

ODIN files encoded in latin 1 (ISO-8859-1) or another variant of ISO-8859 - both containing accented characters with unicode codes outside the ASCII 0-127 range - may work perfectly well, for various reasons:

- the contain nothing but ASCII, i.e. unicode code-points 0 - 127; this will be the case in English-language authored archetypes containing no foreign words;
- some layer of the operating system is smart enough to do an on-the-fly conversion of characters above 127 into UTF-8, even if the archetype tool being used is designed for pure UTF-8 only;
- a tool using the ODIN file might support UTF-8 and ISO-8859 variants.

For situations where binary UTF-8 cannot be supported, ASCII encoding of unicode characters above code-point 127 should only be done using the system supported by many programming languages today, namely \u escaped UTF-16. In this system, unicode codepoints are mapped to either:

- \uHHHH - 4 hex digits which will be the same (possibly 0-filled on the left) as the unicode code-point number expressed in hexadecimal; this applies to unicode codepoints in the range U+0000 - U+FFFF (the 'base multi-lingual plane', BMP);
- \uHHHHHHHH - 8 hex digits to encode unicode code-points in the range U+10000 through U+10FFFF (non-BMP planes); the algorithm is described in IETF RFC 2781[1].

It is not expected that the above approach will be commonly needed, and it may not be needed at all; it is preferable to find ways to ensure that native UTF-8 can be supported, since this reduces the burden for ODIN parser and tool implementers. The above guidance is therefore provided only to ensure a standard approach is used for ASCII-encoded unicode, if it becomes unavoidable.

Thus, while **the only officially designated encoding for ODIN and its constituent syntaxes is UTF-8**, real software systems may be more tolerant. This document therefore specifies that any tool designed to process ODIN files need only support UTF-8; supporting other encodings is an optional extra. This could change in the future, if required by the ODIN user community.

URIs, which have their own data type in ODIN, are handled in a specific way: all characters outside the 'unreserved set' defined by RFC 3986[2] are 'percent-encoded'. The unreserved set is:

```
unreserved  = ALPHA / DIGIT / "-" / "." / "_" / "~"
```

3.2 Special Character Sequences

In string and character data values, characters not in the lower ASCII (0-127) range should normally be UTF-8 encoded, but with the option of using the following quoted forms customary in software development:

1. see http://tools.ietf.org/html/rfc2781.
2. Uniform Resource Identifier (URI): Generic Syntax, Internet proposed standard, January 2005; see http://www.ietf.org/rfc/rfc3986.txt

© 2013- The openEHR Foundation
Issue tracker: www.openehr.org/issues/browse/SPECPR Web: www.openEHR.org

- \r - carriage return
- \n - linefeed
- \t - tab
- \\ - backslash
- \" - literal double quote within a String terminal value
- \' - literal single quote within a Character terminal value

Any other character combination starting with a backslash is illegal; to get the effect of a literal backslash, the \\ sequence should always be used.

Typically in a normal string, including multi-line paragraphs as used in ODIN, only \\ and \" are likely to be necessary, since all of the others can be accommodated in their literal forms; the same goes for single characters - only \\ and \' are likely to commonly occur. However, some authors may prefer to use \n and \t to make intended formatting clearer, or to allow for text editors that do not react properly to such characters. Parsers should therefore support the above list.

3.3 Keywords

ODIN has no keywords of its own: all identifiers are assumed to come from an information model.

3.4 Reserved Characters

In ODIN, a small number of characters are reserved and have the following meanings:

'<': open an object block;

'>': close an object block;

'=': indicate attribute value = object block;

'(', ')': type name or plug-in syntax type delimiters;

'<#': open an object block expressed in a plug-in syntax;

'#>': close an object block expressed in a plug-in syntax.

Within <> delimiters, various characters are used as follows to indicate primitive values:

'"': double quote characters are used to delimit string values;

''': single quote characters are used to delimit single character values;

'|': bar characters are used to delimit intervals;

[]: brackets are used to delimit coded terms.

3.5 Comments

In an ODIN text, comments are indicated by the characters "--". Multi-line comments are achieved using the "--" leader on each line where the comment continues.

In this document, comments are shown in brown.

3.6 Information Model Identifiers

Two types of identifiers from information models are used in ODIN: type names and attribute names. A **type name** is any identifier with an initial upper case letter, followed by any combination of letters,

© 2013- The openEHR Foundation
Issue tracker: www.openehr.org/issues/browse/SPECPR Web: www.openEHR.org

digits and underscores. A **generic type name** (including nested forms) additionally may include commas and angle brackets, and must be syntactically correct as per the UML. An **attribute name** is any identifier with an initial lower case letter, followed by any combination of letters, digits and underscores. Any convention that obeys this rule is allowed.

At least two well-known conventions that are ubiquitous in information models obey the above rule. One of these is the following convention:

- type names are in all uppercase, e.g. `PERSON`, except for 'built-in' types, such as primitive types (`Integer`, `String`, `Boolean`, `Real`, `Double`) and assumed container types (`List<T>`, `Set<T>`, `Hash<T, U>`), which are in mixed case, in order to provide easy differentiation of built-in types from constructed types defined in the reference model. Built-in types are the same types assumed by UML, OCL, IDL and other similar object-oriented formalisms.
- attribute names are shown in all lowercase, e.g. `home_address`.
- in both type names and attribute names, underscores are used to represent word breaks. This convention is used to maximise the readability of this document.

The other common style is the programmer's mixed-case or "camel case" convention exemplified by `Person` and `homeAddress`, as long as they obey the rule above. The convention chosen for any particular ODIN document should be based on the convention used in the underlying information model. Identifiers are shown in green in this document.

3.7 Semi-colons

Semi-colons can be used to separate ODIN blocks, for example when it is preferable to include multiple attribute/value pairs on one line. Semi-colons make no semantic difference at all, and are included only as a matter of taste. The following examples are equivalent:

```
term = <text = <"plan">; description = <"The clinician's advice">>

term = <text = <"plan"> description = <"The clinician's advice">>

term = <
    text = <"plan">
    description = <"The clinician's advice">
>
```

Semi-colons are completely optional in ODIN.

© 2013- The openEHR Foundation
Issue tracker: www.openehr.org/issues/browse/SPECPR Web: www.openEHR.org

4 ODIN Artefacts

An ODIN text may be created in two different physical forms: *embedded fragments* and *documents*. For both the following general structure applies:

```
[ schema_identifier ] main_text
```

The optional `schema_identifier` line takes the following form:

```
@schema = <URI>
```

where 'URI' is an value of the URI primitive type. This is used to indicate the schema, including its version, on which the main ODIN text is based. It is optional because in many cases the schema is known *a priori*, or can be inferred from context.

The following sections describe various types of ODIN artefact.

4.1 Embedded Fragment

Fragments of ODIN text can appear embedded within other artefacts. A fragment typically includes no object identiers nor schema identifier since both of these are assumed to be inferred from the surrounding context. A typical fragment has the following appearance:

```
.... other formalism text ....
.... delimiter ...
--
-- ODIN Embedded Fragment
--
attr_1 = <
    attr_12 = <
       attr_13 = <leaf_value>
    >
>
attr_2 = <
    attr_22 = <leaf_value>
>
.... delimiter ....
.... other formalism text ...
```

4.2 Document

An ODIN document is considered a standalone artefact whose contents can take various forms, all assumed to correspond to one or more serialised objects.

4.2.1 Implicit Object Document

A document containing only an embedded fragment such as shown above is considered to be an 'implict' object document, and its contents are assumed to consist of values of various object properties. This format is a degenerate form of the 'anonymous' object document but so common and useful it is treated as a legal ODIN form.

Implicit object documents are commonly used to serialise models, such as information model schemas, application configuration files and so on. The usual assumption is that the filename and/or ODIN content will identify the artefact sufficiently for tools to know what its information model is.

© 2013- The openEHR Foundation
Issue tracker: www.openehr.org/issues/browse/SPECPR Web: www.openEHR.org

4.2.2 Anonymous Object Document

Any other kind of ODIN document contains one or more explicit objects. The anonymous object form has one object per document, with no object identifier and consists of an outer '<>' delimiter pair containing an ODIN embedded fragment, i.e.:

```
--
-- ODIN Anonymous Object Document
--
<
    attr_1 = <
       attr_12 = <
          attr_13 = <leaf_value>
       >
    >
    attr_2 = <
       attr_22 = <leaf_value>
    >
>
```

This form has no practical benefit over the implicit document form, but is syntactically more correct, and should be supported by parsers.

4.2.3 Identified Object Document

The next variant corresponds to serialisations of multiple objects, each of which is identified.

```
--
-- ODIN Identified Object Document
--
[id_1] = <
    attr_1 = <
       attr_12 = <
          attr_13 = <leaf_value>
       >
    >
>
[id_2] = <
    attr_1 = <
       attr_12 = <
          attr_13 = <leaf_value>
       >
    >
>
...
[id_N] = <
    attr_1 = <
       attr_12 = <
          attr_13 = <leaf_value>
       >
    >
>
```

Identifiers can be values of the String, Integer or any Date/Time primitive types. Strings are most commonly used e.g.:

```
--
-- ODIN Identified Object Document
--
```


© 2013- The openEHR Foundation
Issue tracker: www.openehr.org/issues/browse/SPECPR Web: www.openEHR.org

```
["aaa"] = <
    ...
>
["bbb"] = <
    ...
>
["ccc"] = <
    ...
>
```

Identified Object Documents are most commonly used for representing serialised in-memory instance networks, i.e. the notion of an ‘object dump’.

© 2013- The openEHR Foundation
Issue tracker: www.openehr.org/issues/browse/SPECPR Web: www.openEHR.org

5 Content

5.1 General Structure

Within any kind of ODIN object instance (i.e. implied, anonymous or identified), the structure is a hierarchy of attribute names and object values. In its simplest form, an ODIN object consists of repetitions of the following pattern:

```
attribute_name = <value>
```

Each attribute name is the name of an attribute in an implied or actual object or relational model. Each "value" is either a literal value of a primitive type (see Primitive Types on page 23) or a further nesting of attribute names and values, terminating in leaf nodes of primitive type values. Where sibling attribute nodes occur, the attribute identifiers must be unique, just as in a standard object or relational model.

The following shows a typical structure.

```
attr_1 = <
    attr_2 = <
       attr_3 = <leaf_value>
       attr_4 = <leaf_value>
    >
    attr_5 = <
       attr_3 = <
          attr_6 = <leaf_value>
       >
       attr_7 = <leaf_value>
    >
>
attr_8 = <>
```

In the above structure, each "<>" encloses an instance of some type. The hierarchical structure corresponds to the part-of relationship between objects, otherwise known as *composition* and *aggregation* relationships in object-oriented formalisms such as UML (the difference between the two is usually described as being "sub-objects related by aggregation can have independent lifetimes, whereas sub-objects related by composition have co-termimal lifetimes and are always destroyed with the parent"; ODIN does not differentiate between the two, since it is the business of a model, not the data, to express such semantics). Associations between instances in ODIN are also representable by references, and are described in section 5.7 on page 21.

Validity rules for object-structuring of ODIN are as follows:

> **VDATU: attribute name uniqueness**: sibling attributes occurring within an object node must be uniquely named with respect to each other, in the same way as in class definitions in an information model.

5.2 Paths

For any ODIN structure, a set of paths can be extracted that correspond to the tree structure of the data. The complete set of paths for the above example is as follows.

```
/attr_1
/attr_1/attr_2
/attr_1/attr_2/attr_3       -- path to a leaf value
```


© 2013- The openEHR Foundation
Issue tracker: www.openehr.org/issues/browse/SPECPR Web: www.openEHR.org

```
/attr_1/attr_2/attr_4       -- path to a leaf value
/attr_1/attr_5
/attr_1/attr_5/attr_3
/attr_1/attr_5/attr_3/attr_6 -- path to a leaf value
/attr_1/attr_5/attr_7       -- path to a leaf value
/attr_8
```

The path syntax used with ODIN maps trivially to W3C Xpath and Xquery paths, and is described in section 7.

5.3 Void Objects

A *void object*, i.e. an object attribute that has no value is allowed in an ODIN text, but ignored by parsers. It is legal to output void objects, but not recommended. A void object looks as follows:

```
address = <>          -- person's address
```

5.4 Container Objects

The syntax described so far allows an instance of an arbitrarily large object to be expressed, but does not support attributes of container types such as lists, sets and hash tables, i.e. items whose type in an underlying reference model is something like `attr:List<Type>`, `attr:Set<Type>` or `attr:Hash<ValueType, KeyType>`. There are two ways instance data of such container objects can be expressed in ODIN. The first applies to leaf values and is to use a list style literal value for , where the "list nature" of the data is expressed within the manifest value itself, as in the following examples.

```
fruits = <"pear", "cumquat", "peach">
some_primes = <1, 2, 3, 5>
```

See Lists of Built-in Types on page 26 for the complete description of list leaf types.

However for containers holding non-primitive values, including more container objects, a different syntax is needed. Consider by way of example that an instance of the container `List<Person>` could be expressed as follows.

```
-- WARNING: THIS IS NOT VALID ODIN
people = <
    <name = <...> date_of_birth = <...> sex = <> interests = <...>>
    <name = <...> date_of_birth = <...> sex = <> interests = <...>>
    -- etc
>
```

Here, 'anonymous' blocks of data are repeated inside the outer block. However, this makes the data hard to read, and does not provide an easy way of constructing paths to the contained items. A better syntax becomes more obvious when we consider that members of container objects in their computable form are nearly always accessed by a method such as `member(i)`, `item[i]` or just plain `[i]`, in the case of array access in the C-based languages.

ODIN opts for the array-style syntax, known in ODIN as container member *keys*. No attribute name is explicitly given; any primitive comparable value is allowed as the key, rather than just integers used in C-style array access. Further, if integers are used, it is not assumed that they dictate ordinal indexing, i.e. it is possible to use a series of keys `[2]`, `[4]`, `[8]` etc. The following example shows one version of the above container in valid ODIN:

```
people = <
    [1] = <name = <...> birth_date = <...> interests = <...>>
    [2] = <name = <...> birth_date = <...> interests = <...>>
```


© 2013- The openEHR Foundation
Issue tracker: www.openehr.org/issues/browse/SPECPR Web: www.openEHR.org

```
        [3] = <name = <…> birth_date = <…> interests = <…>>
    >
```

Strings and dates may also be used. Keys are coloured blue in the this specification in order to distinguish the run-time status of key values from the design-time status of class and attribute names. The following example shows the use of string values as keys for the contained items.

```
people = <
    ["akmal:1975-04-22"] = <name = <…> birth_date = <…> interests = <…>>
    ["akmal:1962-02-11"] = <name = <…> birth_date = <…> interests = <…>>
    ["gianni:1978-11-30"] = <name = <…> birth_date = <…> interests = <…>>
>
```

The syntax for primitive values used as keys follows exactly the same syntax described below for data of primitive types. It is convenient in some cases to construct key values from one or more of the values of the contained items, in the same way as relational database keys are constructed from sufficient field values to guarantee uniqueness. However, they need not be - they may be independent of the contained data, as in the case of hash tables, where the keys are part of the hash table structure, or equally, they may simply be integer index values, as in the 'locations' attribute in the 'school_schedule' structure shown below.

Container structures can appear anywhere in an overall instance structure, allowing complex data such as the following to be expressed in a readable way.

```
school_schedule = <
    lesson_times = <08:30:00, 09:30:00, 10:30:00, ...>

    locations = <
        [1] = <"under the big plane tree">
        [2] = <"under the north arch">
        [3] = <"in a garden">
    >

    subjects = <
        ["philosophy:plato"] = < -- note construction of key
            name = <"philosophy">
            teacher = <"plato">
            topics = <"meta-physics", "natural science">
            weighting = <76%>
        >
        ["philosophy:kant"] = <
            name = <"philosophy">
            teacher = <"kant">
            topics = <"meaning and reason", "meta-physics", "ethics">
            weighting = <80%>
        >
        ["art"] = <
            name = <"art">
            teacher = <"goya">
            topics = <"technique", "portraiture", "satire">
            weighting = <78%>
        >
    >
```

The example above conforms directly to the object-oriented type specification (given in a pascal-like syntax):

```
class SCHEDULE
```


© 2013- The openEHR Foundation
Issue tracker: www.openehr.org/issues/browse/SPECPR Web: www.openEHR.org

```
    lesson_times: List<Time>
    locations: List<String>
    subjects: List<SUBJECT> -- or it could be Hash<SUBJECT>
end

class SUBJECT
    name: String
    teacher: String
    topics: List<String>
    weighting: Real
end
```

Other class specifications corresponding to the same data are possible, but will all be isomorphic to the above.

How key values relate to a particular object structure depends on the object model being used during the ODIN parsing process. It is possible to write a parser which makes reasonable inferences from an information model whose instances are represented as ODIN text; it is also possible to include explicit typing information in the ODIN itself (see Adding Type Information below).

The validity rule for objects within a container attribute is as follows:

> **VDOBU: object identifier uniqueness**: sibling objects occurring within a container attribute must be uniquely identified with respect to each other.

Paths through container objects are formed in the same way as paths in other structured data, with the addition of the key, to ensure uniqueness. The key is included syntactically enclosed in brackets, in a similar way to Xpath predicates. Paths through containers in the above example include the following:

```
/school_schedule/locations[1]                -- path to "under the big..."
/school_schedule/subjects["philosophy:kant"]     -- path to "kant"
```

5.5 Nested Container Objects

In some cases the data of interest are instances of nested container types, such as `List<List<Message>>` (a list of Message lists) or `Hash<List<Integer>, String>` (a hash of integer lists keyed by strings). The ODIN syntax for such structures follows directly from the syntax for a single container object. The following example shows an instance of the type `List<List<String>>` expressed in ODIN syntax.

```
list_of_string_lists = <
    [1] = <
        [1] = <"first string in first list">
        [2] = <"second string in first list">
    >
    [2] = <
        [1] = <"first string in second list">
        [2] = <"second string in second list">
        [3] = <"third string in second list">
    >
    [3] = <
        [1] = <"only string in third list">
    >
>
```

The paths of the above example are as follows:

```
/list_of_string_lists[1]/[1]
```


© 2013- The openEHR Foundation
Issue tracker: www.openehr.org/issues/browse/SPECPR Web: www.openEHR.org

```
/list_of_string_lists[1]/[2]
/list_of_string_lists[2]/[1]
etc
```

5.6 Adding Type Information

In many cases, ODIN data is of a simple structure, very regular, and highly repetitive, such as the expression of simple demographic data. In such cases, it is preferable to express as little as possible about the informatoin model on which the data are based, since various software components want to use the data, and use it in different ways. However, there are also cases where the data is highly complex, and more model information is needed to help a parser. Examples include large design databases for aircraft and health records. Data obeying more complex models typically include sub-objects that are of a subtype of the statically declared type in the information model, in other words, dynamically bound types.

Where dynamic binding occurs in the data, it must be indicated in anODIN document. Typing information is added to using a syntactical addition inspired by the `(type)` casting operator of the C language, whose meaning is approximately: force the type of the result of the following expression to be `type`. In ODIN typing is therefore done by including the type name in parentheses after the '=' sign, as in the following example.

```
destinations = <
    ["seville"] = (TOURIST_DESTINATION) <
        profile = (DESTINATION_PROFILE) <...>
        hotels = <
            ["gran sevilla"] = (HISTORIC_HOTEL) <...>
            ["sofitel"] = (LUXURY_HOTEL) <...>
            ["hotel real"] = (PENSION) <...>
        >
        attractions = <
            ["la corrida"] = (SPORT_VENUE) <...>
            ["Alcázar"] = (HISTORIC_SITE) <...>
        >
    >
>
```

The path set from the above example is as follows:

```
/destinations["seville"]/hotels["gran sevilla"]
/destinations["seville"]/hotels["sofitel"]
/destinations["seville"]/hotels["hotel real"]

/bookings["seville:0134"]/customer_id
/bookings["seville:0134"]/period
/bookings["seville:0134"]/hotel

/hotels["sofitel"]
/hotels["hotel real"]
/hotels["gran sevilla"]
```

In the above, no type identifiers are included after the `hotels` and `attractions` attributes, so it is assumed by the parser that they are of their statically declared type, typically something like `List<HOTEL>` and `List<ATTRACTION>` respectively. Nevertheless, complete typing information can be included, as follows.

```
hotels = (List<HOTEL>) <
    ["gran sevilla"] = (HISTORIC_HOTEL) <...>
>
```


© 2013- The openEHR Foundation
Issue tracker: www.openehr.org/issues/browse/SPECPR Web: www.openEHR.org

This illustrates the use of generic, i.e. template types, expressed in the standard UML syntax, using angle brackets. Any number of template arguments and any level of nesting is allowed, as in the UML. At first view, there may appear to be a risk of confusion between template type '<>' delimiters and the standard ODIN block delimiters. However the parsing rules are easy to state; essentially the difference is that an ODIN data block is always preceded by an '=' symbol.

Type identifiers can also include namespace information, which is necessary when same-named types appear in different packages of a model. Namespaces are included by prepending package names, separated by the '.' character, in the same way as in most programming languages, as in the qualified type names **org.openehr.rm.ehr.content.ENTRY** and **Core.Abstractions.Relationships.Relationship**.

5.7 Associations and Shared Objects

All of the facilities described so far allow any object-oriented data to be faithfully expressed in a formal, systematic way which is both machine- and human-readable, and allow any node in the data to be addressed using an Xpath-style path. The availability of reliable paths allows not only the representation of single 'business objects', which are the equivalent of UML *aggregation* (and *composition*) hierarchies, but also the representation of *associations* between objects, and by extension, shared objects.

5.7.1 Within An Object

Consider that in the example above, 'hotel' objects may be shared objects, referred to by assocation. This can be expressed as follows.

```
destinations = <
    ["seville"] = <
        hotels = <
            ["gran sevilla"] = </hotels["gran sevilla"]>
            ["sofitel"] = </hotels["sofitel"]>
            ["hotel real"] = </hotels["hotel real"]>
        >
    >
>

bookings = <
    ["seville:0134"] = <
        customer_id = <"0134">
        period = <...>
        hotel = </hotels["sofitel"]>
    >
>

hotels = <
    ["gran sevilla"] = (HISTORIC_HOTEL) <...>
    ["sofitel"] = (LUXURY_HOTEL) <...>
    ["hotel real"] = (PENSION) <...>
>
```

Associations are expressed via the use of fully qualified paths as the data for a attribute. In this example, there are references from a list of destinations, and from a booking list, to the same hotel object. If type information is included, it should go in the declarations of the relevant objects; type declara-

© 2013- The openEHR Foundation
Issue tracker: www.openehr.org/issues/browse/SPECPR Web: www.openEHR.org

tions can also be used before path references, which might be useful if the association type is an ancestor type (i.e. more general type) of the type of the actual object being referred to.

5.7.2 Across Objects

In an ODIN document containing identified objects, with references across objects, reference paths will include object identifiers, as shown below:

```
["travel_db_0293822"] = <
    destinations = <
        ["seville"] = <
            hotels = <
                ["gran sevilla"] = <["tourism_db_13"]/hotels["gran sevilla"]>
                ["sofitel"] = <["tourism_db_13"]/hotels["sofitel"]>
                ["hotel real"] = <["tourism_db_13"]/hotels["hotel real"]>
            >
        >
    >

    bookings = <
        ["seville:0134"] = <
            customer_id = <"0134">
            period = <...>
            hotel = <["tourism_db_13"]/hotels["sofitel"]>
        >
    >
>

["tourism_db_13"] = <
    hotels = <
        ["gran sevilla"] = (HISTORIC_HOTEL) <...>
        ["sofitel"] = (LUXURY_HOTEL) <...>
        ["hotel real"] = (PENSION) <...>
    >
>
```

5.7.3 Across ODIN Documents

Data in other ODIN documents can be referred to using a URI containing a reference path to locate the document, with the internal path included as described above.

© 2013- The openEHR Foundation
Issue tracker: www.openehr.org/issues/browse/SPECPR Web: www.openEHR.org

6 Leaf Data

All ODIN data eventually devolve to instances of the primitive types `String`, `Integer`, `Real`, `Double`, `String`, `Character`, various date/time types, lists or intervals of these types, and a few special types. ODIN does not use type or attribute names for instances of primitive types, only manifest values, making it possible to assume as little as possible about type names and structures of the primitive types. In all the following examples, the manifest data values are assumed to appear immediately inside a leaf pair of angle brackets, i.e.

```
some_attribute = <manifest value>
```

6.1 Primitive Types

6.1.1 Character Data

Characters are shown in a number of ways. In the literal form, a character is shown in single quotes, as follows:

```
'a'
```

Characters outside the low ASCII (0-127) range must be UTF-8 encoded, with a small number of backslash-quoted ASCII characters allowed, as described in File Encoding on page 10.

6.1.2 String Data

All strings are enclosed in double quotes, as follows:

```
"this is a string"
```

Quotes are encoded using ISO/IEC 10646 codes, e.g. :

```
"this is a much longer string, what one might call a &quot;phrase&quot;."
```

Line extension of strings is done simply by including returns in the string. The exact contents of the string are computed as being the characters between the double quote characters, with the removal of white space leaders up to the left-most character of the first line of the string. This has the effect of allowing the inclusion of multi-line strings in ODIN texts, in their most natural human-readable form, e.g.:

```
text = <"And now the STORM-BLAST came, and he
        Was tyrannous and strong :
        He struck with his o'ertaking wings,
        And chased us south along.">
```

String data can be used to contain almost any other kind of data, which is intended to be parsed as some other formalism. Characters outside the low ASCII (0-127) range must be UTF-8 encoded, with a small number of backslash-quoted ASCII characters allowed, as described in File Encoding on page 10.

6.1.3 Integer Data

Integers are represented simply as numbers, e.g.:

```
25
300000
29e6
```

Commas or periods for breaking long numbers are not allowed, since they confuse the use of commas used to denote list items (see section 6.4 below).

© 2013- The openEHR Foundation
Issue tracker: www.openehr.org/issues/browse/SPECPR Web: www.openEHR.org

6.1.4 Real Data

Real numbers are assumed whenever a decimal is detected in a number, e.g.:

```
25.0
3.1415926
6.023e23
```

Commas or periods for breaking long numbers are not allowed. Only periods may be used to separate the decimal part of a number; unfortunately, the European use of the comma for this purpose conflicts with the use of the comma to distinguish list items (see section 6.4 below).

6.1.5 Boolean Data

Boolean values can be indicated by the following values (case-insensitive):

```
True
False
```

6.1.6 Dates and Times

Complete Date/Times

In ODIN, full and partial dates, times and durations can be expressed. All full dates, times and durations are expressed using a subset of ISO8601. The Support IM provides a full explanation of the ISO8601 semantics supported in openEHR.

In ODIN, the use of ISO 8601 allows extended form only (i.e. ':' and '-' must be used). The ISO 8601 method of representing partial dates consisting of a single year number, and partial times consisting of hours only are not supported, since they are ambiguous. See below for partial forms.

Patterns for complete dates and times in ODIN include the following:

```
yyyy-MM-dd                          -- a date
hh:mm:ss[,sss][Z|+/-hhmm]           -- a time with optional seconds
yyyy-MM-ddThh:mm:ss[,sss][Z]        -- a date/time
```

where:

```
yyyy    = four-digit year
MM      = month in year
dd      = day in month
hh      = hour in 24 hour clock
mm      = minutes
ss,sss  = seconds, incuding fractional part
Z       = the timezone in the form of a '+' or '-' followed by 4 digits
          indicating the hour offset, e.g. +0930, or else the literal 'Z'
          indicating +0000 (the Greenwich meridian).
```

Durations are expressed using a string which starts with 'P', and is followed by a list of periods, each appended by a single letter designator: 'Y' for years, "M' for months, 'W' for weeks, 'D' for days, 'H' for hours, 'M' for minutes, and 'S' for seconds. The literal 'T' separates the YMWD part from the HMS part, ensuring that months and minutes can be distinguished. Examples of date/time data include:

```
1919-01-23              -- birthdate of Django Reinhardt
16:35:04,5              -- rise of Venus in Sydney on 24 Jul 2003
2001-05-12T07:35:20+1000 -- timestamp on an email received from Australia
P22D4TH15M0S            -- period of 22 days, 4 hours, 15 minutes
```


© 2013- The openEHR Foundation
Issue tracker: www.openehr.org/issues/browse/SPECPR Web: www.openEHR.org

Partial Date/Times

Two ways of expressing partial (i.e. incomplete) date/times are supported in ODIN. The ISO 8601 incomplete formats are supported in extended form only (i.e. with '-' and ':' separators) for all patterns that are unambiguous on their own. Dates consisting of only the year, and times consisting of only the hour are not supported, since both of these syntactically look like integers. The supported ISO 8601 patterns are as follows:

```
yyyy-MM                         -- a date with no days
hh:mm                           -- a time with no seconds
yyyy-MM-ddThh:mm                -- a date/time with no seconds
yyyy-MM-ddThh                   -- a date/time, no minutes or seconds
```

To deal with the limitations of ISO 8601 partial patterns in a context-free parsing environment, a second form of pattern is supported in ODIN, based on ISO 8601. In this form, '?' characters are substituted for missing digits. Valid partial dates follow the patterns:

```
yyyy-MM-??              -- date with unknown day in month
yyyy-??-??              -- date with unknown month and day
```

Valid partial times follow the patterns:

```
hh:mm:??                -- time with unknown seconds
hh:??:??                -- time with unknown minutes and seconds
```

Valid date/times follow the patterns:

```
yyyy-MM-ddThh:mm:??     -- date/time with unknown seconds
yyyy-MM-ddThh:??:??     -- date/time with unknown minutes and seconds
yyyy-MM-ddT??:??:??     -- date/time with unknown time
yyyy-MM-??T??:??:??     -- date/time with unknown day and time
yyyy-??-??T??:??:??     -- date/time with unknown month, day and time
```

6.2 Intervals of Ordered Primitive Types

Intervals of any ordered primitive type, i.e., Integer, Real, Date, Time, Date_time and Duration, can be stated using the following uniform syntax, where N, M are instances of any of the ordered types:

`|N..M|`	the two-sided range N >= x <= M;
`|N>..M|`	the two-sided range N > x <= M;
`|N..<M|`	the two-sided range N >= x <M;
`|N>..<M|`	the two-sided range N > x <M;
`|<N|`	the one-sided range x < N;
`|>N|`	the one-sided range x > N;
`|>=N|`	the one-sided range x >= N;
`|<=N|`	the one-sided range x <= N;
`|N +/-M|`	interval of N ± M.

The allowable values for N and M include any value in the range of the relevant type, as well as:

```
infinity
-infinity
```

* equivalent to `infinity`

Examples of this syntax include:

```
|0..5|                  -- integer interval
|0.0..1000.0|            -- real interval
```


© 2013- The openEHR Foundation
Issue tracker: www.openehr.org/issues/browse/SPECPR Web: www.openEHR.org

```
|0.0..<1000.0|          -- real interval 0.0 >= x < 1000.0
|08:02..09:10|          -- interval of time
|>= 1939-02-01|         -- open-ended interval of dates
|5.0 +/-0.5|            -- 4.5 - 5.5
|>=0|                   -- >= 0
|0..infinity|           -- 0 - infinity (i.e. >= 0)
```

6.3 Other Built-in Types

6.3.1 URIs

URI can be expressed as ODIN data in the usual way found on the web, and follow the standard syntax from http://www.ietf.org/rfc/rfc3986.txt. Examples of URIs in ODIN:

```
http://openEHR.org/home
ftp://get.this.file.com?file=cats.doc#section_5
http://www.mozilla.org/products/firefox/upgrade/?application=thunderbird
```

Encoding of special characters in URIs follows the IETF RFC 3986, as described under File Encoding on page 10.

6.3.2 Coded Terms

Coded terms are ubiquitous in medical and clinical information, and are likely to become so in most other industries, as ontologically-based information systems and the ‘semantic web’ emerge. The logical structure of a coded term is simple: it consists of an identifier of a terminology (with optional version), and an identifier of a code within that terminology. The ODIN string representation is of the following form:

```
[terminology_id::code]
```

where `terminology_id` is an alphanumeric name, optionally following by a version in parentheses, and `code` is a string. The allowed characters in each part are described in the grammaar.

Examples from clinical data:

```
[icd10AM::F60.1]            -- from ICD10AM
[snomed_ct::2004950]        -- from snomed-ct
[snomed_ct(3.1)::2004950]   -- from snomed-ct v 3.1
```

6.4 Lists of Built-in Types

Data of any primitive type can occur singly or in lists, which are shown as comma-separated lists of item, all of the same type, such as in the following examples:

```
“cyan”, “magenta”, “yellow”, “black” -- printer’s colours
1, 1, 2, 3, 5                        -- first 5 fibonacci numbers
08:02, 08:35, 09:10                  -- set of train times
```

No assumption is made in the syntax about whether a list represents a set, a list or some other kind of sequence - such semantics must be taken from an underlying information model.

Lists which happen to have only one datum are indicated by using a comma followed by a list continuation marker of three dots, i.e. “...”, e.g.:

```
“en”, ...       -- languages
“icd10”, ...    -- terminologies
[at0200], ...
```

White space may be freely used or avoided in lists, i.e. the following two lists are identical:

```
1,1,2,3
```


© 2013- The openEHR Foundation
Issue tracker: www.openehr.org/issues/browse/SPECPR Web: www.openEHR.org

```
1, 1, 2,3
```


© 2013- The openEHR Foundation
Issue tracker: www.openehr.org/issues/browse/SPECPR Web: www.openEHR.org

7 Path Syntax

7.1 Semantics

The general form of the path syntax is as follows (see syntax section below for full specification):

```
path: ['/'] path_segment { '/' path_segment }+
path_segment: attr_name [ '[' object_id ']' ]
```

Essentially, ODIN paths consist of segments separated by slashes ('/'), where each segment is an attribute name with optional object identifier predicate, indicated by brackets ('[]').

ODIN Paths are formed from an alternation of segments made up of an attribute name and optional object node identifier predicate, separated by slash ('/') characters. Node identifiers are delimited by brackets (i.e. []).

Similarly to paths used in file systems, ODIN paths are either absolute or relative, with the former being indicated by a leading slash.

Paths are *absolute* or *relative* with respect to the document in which they are mentioned. Absolute paths commence with an initial slash ('/') character.

TBD_2: The ODIN path syntax also supports the concept of "movable" path patterns, i.e. paths that can be used to find a section anywhere in a hierarchy that matches the path pattern. Path patterns are indicated with a leading double slash ("//") as in Xpath.

Path *patterns* are absolute or relative with respect to the document in which they are mentioned. Absolute paths commence with an initial slash ('/') character.

A typical ODIN path used to refer to a node in an ODIN text is as follows.

```
/term_definitions["en"]/items["at0001"]/text
```

In the following sections, paths are shown for all the ODIN data examples.

7.2 Relationship with W3C Xpath

The ODIN path syntax is semantically a subset of the Xpath query language, with a few syntactic shortcuts to reduce the verbosity of the most common cases. Xpath differentiates between "children" and "attributes" sub-items of an object due to the difference in XML between Elements (true sub-objects) and Attributes (tag-embedded primitive values). In ODIN, as with any pure object formalism, there is no such distinction, and all subparts of any object are referenced in the manner of Xpath children; in particular, in the Xpath abbreviated syntax, the key `child::` does not need to be used.

ODIN does not distinguish attributes from children, and also assumes the `node_id` attribute. Thus, the following expressions are legal for cADL structures:

```
items[1]            -- the first member of 'items'
items["systolic"]   -- the member of 'items' with meaning 'systolic'
items["at0001"]     -- the member of 'items' with node id 'at0001'
```

The Xpath equivalents are:

```
items[1]            -- the first member of 'items'
items[@key = 'systolic'] -- the member of 'items' with key "systolic"
```


© 2013- The openEHR Foundation
Issue tracker: www.openehr.org/issues/browse/SPECPR Web: www.openEHR.org

```
items[@archetype_node_id = 'at0001']
        -- the member of 'items' with archetype_node_id attribute 'at0001'
```


© 2013- The openEHR Foundation
Issue tracker: www.openehr.org/issues/browse/SPECPR Web: www.openEHR.org

8 Plug-in Syntaxes

Using the ODIN syntax, any object structure can be serialised. In some cases, the requirement is to express some part of the structure in an abstract syntax, rather than in the more literal seriliased object form of ODIN. ODIN provides for this possibility by allowing the value of any object (i.e. what appears between any matching pair of <> delimiters) to be expressed in some other syntax, known as a “plug-in” syntax. Plug-in syntaxes are indicated in ODIN in a similar way as typed objects, i.e. by the use of the syntax type in parentheses preceding the <> block. For a plug-in section, the <> delimiters are modified to <# #>, to allow for easier parser design, and easier recognition of such blocks by human readers. The general form is as follows:

```
attr_name = (syntax) <#
    ...
#>
```

The following example illustrates a cADL plug-in section in an archetype, which it itself an ODIN document:

```
definition = (cadl) <#
    ENTRY[at0000] ∈ {          -- blood pressure measurement
       name ∈ {                -- any synonym of BP
          CODED_TEXT ∈ {
             code ∈ {
                CODE_PHRASE ∈ {[ac0001]}
             }
          }
       }
    }
#>
```

Clearly, many plug-in syntaxes might one day be used within ODIN data; there is no guarantee that every ODIN parser will support them. The general approach to parsing should be to use plug-in parsers, i.e. to obtain a parser for a plug-in syntax that can be built into the existing parser framework.

© 2013- The openEHR Foundation
Issue tracker: www.openehr.org/issues/browse/SPECPR Web: www.openEHR.org

9 Relationship with other Syntaxes

9.1 XML

A common question about ODIN is why it is needed, when there is already XML? To start with, this question highlights the widespread misconception about XML, namely that because it can be read by a text editor, it is intended for humans. In fact, XML is designed for machine processing, and is textual to guarantee its interoperability, not its readability. Realistic examples of XML (e.g. XML-schema instance, OWL-RDF ontologies) are generally unreadable for humans. ODIN is on the other hand designed as a human-writable and readable formalism that is also machine processable; it may be thought of as an *abstract syntax for object-oriented data*. ODIN also differs from XML by:

- providing a more comprehensive set of leaf data types, including intervals of numerics and date/time types, and lists of all primitive types;
- adhering to object-oriented semantics, particularly for container types, which XML schema languages generally do not;
- not using the confusing XML notion of 'attributes' and 'elements' to represent what are essentially object properties;
- requiring roughly half the space of the equivalent XML.

Of course, this does not prevent XML exchange syntaxes being used for ODIN, and indeed the conversion to XML instance is rather straighforward.

9.1.1 Expression of ODIN in XML

The ODIN syntax maps quite easily to XML instance. It is important to realise that people using XML often develop different mappings for object-oriented data, due to the fact that XML does not have systematic object-oriented semantics. This is particularly the case where containers such as lists and sets such as 'employees: List<Person>' are mapped to XML; many implementors have to invent additional tags such as 'employee' to make the mapping appear visually correct. The particular mapping chosen here is designed to be a faithful reflection of the semantics of the object-oriented data, and does not try take into account visual aesthetics of the XML. The result is that Xpath expressions are the same for ODIN and XML, and also correspond to what one would expect based on an underlying object model.

The main elements of the mapping are as follows.

Single Attributes

Single attribute nodes map to tagged nodes of the same name.

Container Attributes

Container attribute nodes map to a series of tagged nodes of the same name, each with the XML attribute 'id' set to the ODIN key. For example, the ODIN:

```
subjects = <
    ["philosophy:plato"] = <
        name = <"philosophy">
    >
    ["philosophy:kant"] = <
        name = <"philosophy">
    >
>
```


© 2013- The openEHR Foundation
Issue tracker: www.openehr.org/issues/browse/SPECPR Web: www.openEHR.org

maps to the XML:

```
<subjects id="philosophy:plato">
    <name>
       philosophy
    </name>
</subjects>
<subjects id="philosophy:kant">
    <name>
       philosophy
    </name>
</subjects>
```

This guarantees that the path `subjects[@id='philosophy:plato']/name` navigates to the same element in both ODIN and the XML.

Nested Container Attributes

Nested container attribute nodes map to a series of tagged nodes of the same name, each with the XML attribute 'id' set to the ODIN key. For example, consider an object structure defined by the signature `countries:Hash<Hash<Hotel,String>,String>`. An instance of this in ODIN looks as follows:

```
countries = <
    ["spain"] = <
       ["hotels"] = <...>
       ["attractions"] = <...>
    >
    ["egypt"] = <
       ["hotels"] = <...>
       ["attractions"] = <...>
    >
>
```

can be mapped to the XML in which the synthesised element tag "_items" and the attribute "key" are used:

```
<countries key="spain">
    <_items key="hotels">
       ...
    </_items>
    <_items key="attractions">
       ...
    </_items>
<countries>
</countries key="egypt">
    <_items id="hotels">
       ...
    </_items>
    <_items key="attractions">
       ...
    </_items>
</countries>
```

In this case, the ODIN path `countries["spain"]/["hotels"]` will be transformed to the Xpath `countries[@key="spain"]/_items[@key="hotels"]` in order to navigate to the same element.

Type Names

Type names map to XML 'type' attributes e.g. the ODIN:

```
destinations = <
```


© 2013- The openEHR Foundation
Issue tracker: www.openehr.org/issues/browse/SPECPR Web: www.openEHR.org

```
        ["seville"] = (TOURIST_DESTINATION) <
            profile = (DESTINATION_PROFILE) <...>
            hotels = <
                ["gran sevilla"] = (HISTORIC_HOTEL) <...>
            >
        >
    >
```

maps to:

```
    <destinations id="seville" adl:type="TOURIST_DESTINATION">
        <profile adl:type="DESTINATION_PROFILE">
            ...
        </profile>
        <hotels id="gran sevilla" adl:type="HISTORIC_HOTEL">
            ...
        </hotels>
    >
>
```

9.2 JSON

The Java Simple Object Notation (JSON) was designed with the aim of representing Java objects in a programming language independent way, primarily for use with the web. The majority of use was for small fragments, although in more recent years it is starting to be used for more complex data representation tasks.

9.2.1 ODIN to JSON Conversion

An ODIN document can be converted into JSON reasonably easily.

© 2013- The openEHR Foundation
Issue tracker: www.openehr.org/issues/browse/SPECPR Web: www.openEHR.org

© 2013- The openEHR Foundation
Issue tracker: www.openehr.org/issues/browse/SPECPR Web: www.openEHR.org

Appendix A Relationship with other Syntaxes

A.1 XML

A common question about ODIN is why it is needed, when there is already XML? To start with, this question highlights the widespread misconception about XML, namely that because it can be read by a text editor, it is intended for humans. In fact, XML is designed for machine processing, and is textual to guarantee its interoperability, not its readability. Realistic examples of XML (e.g. XML-schema instance, OWL-RDF ontologies) are generally unreadable for humans. ODIN is on the other hand designed as a human-writable and readable formalism that is also machine processable; it may be thought of as an *abstract syntax for object-oriented data*. ODIN also differs from XML by:

- providing a more comprehensive set of leaf data types, including intervals of numerics and date/time types, and lists of all primitive types;
- adhering to object-oriented semantics, particularly for container types, which XML schema languages generally do not;
- not using the confusing XML notion of 'attributes' and 'elements' to represent what are essentially object properties;
- requiring roughly half the space of the equivalent XML.

This does not prevent ODIN documents being converted to XML and indeed the conversion to XML instance is rather straighforward.

A.1.1 Expression of ODIN in XML

The ODIN syntax maps relatively easily to XML instance. It is important to realise that developers using XML often develop different mappings for object-oriented data, due to the fact that XML does not have systematic object-oriented semantics. This is particularly the case where containers such as lists and sets such as 'employees: List<Person>' are mapped to XML; many implementors have to invent additional tags such as 'employee' to make the mapping appear visually correct. The particular mapping chosen here is designed to be a faithful reflection of the semantics of the object-oriented data, and does not try take into account visual aesthetics of the XML. The result is that Xpath expressions are the same for ODIN and XML, and also correspond to what one would expect based on an underlying object model.

The main elements of the mapping are as follows.

Single Attributes

Single attribute nodes map to tagged nodes of the same name.

Container Attributes

Container attribute nodes map to a series of tagged nodes of the same name, each with the XML attribute 'id' set to the ODIN key. For example, the ODIN:

```
subjects = <
    ["philosophy:plato"] = <
       name = <"philosophy">
    >
    ["philosophy:kant"] = <
       name = <"philosophy">
    >
>
```


© 2013- The openEHR Foundation
Issue tracker: www.openehr.org/issues/browse/SPECPR Web: www.openEHR.org

maps to the XML:

```
<subjects id="philosophy:plato">
    <name>
       philosophy
    </name>
</subjects>
<subjects id="philosophy:kant">
    <name>
       philosophy
    </name>
</subjects>
```

This guarantees that the path `subjects[@id='philosophy:plato']/name` navigates to the same element in both ODIN and the XML.

Nested Container Attributes

Nested container attribute nodes map to a series of tagged nodes of the same name, each with the XML attribute 'id' set to the ODIN key. For example, consider an object structure defined by the signature `countries:Hash<Hash<Hotel,String>,String>`. An instance of this in ODIN looks as follows:

```
countries = <
    ["spain"] = <
       ["hotels"] = <...>
       ["attractions"] = <...>
    >
    ["egypt"] = <
       ["hotels"] = <...>
       ["attractions"] = <...>
    >
>
```

can be mapped to the XML in which the synthesised element tag "_items" and the attribute "key" are used:

```
<countries key="spain">
    <_items key="hotels">
       ...
    </_items>
    <_items key="attractions">
       ...
    </_items>
<countries>
</countries key="egypt">
    <_items id="hotels">
       ...
    </_items>
    <_items key="attractions">
       ...
    </_items>
</countries>
```

In this case, the ODIN path `countries["spain"]/["hotels"]` will be transformed to the Xpath `countries[@key="spain"]/_items[@key="hotels"]` in order to navigate to the same element.

Type Names

Type names map to XML 'type' attributes e.g. the ODIN:

```
destinations = <
```


© 2013- The openEHR Foundation
Issue tracker: www.openehr.org/issues/browse/SPECPR Web: www.openEHR.org

```
        ["seville"] = (TOURIST_DESTINATION) <
            profile = (DESTINATION_PROFILE) <...>
            hotels = <
                ["gran sevilla"] = (HISTORIC_HOTEL) <...>
            >
        >
    >
```

maps to:

```
    <destinations id="seville" adl:type="TOURIST_DESTINATION">
            <profile adl:type="DESTINATION_PROFILE">
                ...
            </profile>
            <hotels id="gran sevilla" adl:type="HISTORIC_HOTEL">
                ...
            </hotels>
        >
    >
```

A.2 JSON

The Java Simple Object Notation (JSON) was designed with the aim of representing Java objects in a programming language independent way, primarily for use with the web. The majority of use was for small fragments, although in more recent years it is starting to be used for more complex data representation tasks.

A.2.1 Leaf types

ODIN has more terminal types than JSON, including the date/time types, and the Interval types.

Date/time types would typically be mapped to and from Strings containing ISO8601 syntax dates and times.

The interval is a built-in ODIN type that would need to be explicitly expanded into a JSON structure, with an assumed model of the parts of the Interval. For this purpose, the following model is recommended as a basis for constructing the JSON equivalent:

```
class Interval <T: Ordered> {
    T lower;
    T upper;
    Boolean lower_included;
    Boolean upper_included;
}
```

A.2.2 Typing

ODIN supports optional type markers, which are not available with JSON. In a conversion situation these would need to be converted to an explicit structure.

© 2013- The openEHR Foundation
Issue tracker: www.openehr.org/issues/browse/SPECPR Web: www.openEHR.org

© 2013- The openEHR Foundation
Issue tracker: www.openehr.org/issues/browse/SPECPR Web: www.openEHR.org

Appendix B Syntax Specification

B.1 Overview

The grammar and lexical specification for the standard ODIN syntax is shown below. Various implementations exist, and programming resources can be found at the openEHR/ODIN GitHub project.

B.1.1 Production Rules

The following production rules were extracted from the .y file used in the Eiffel ODIN implementation.

```
input:
  attr_vals
| complex_object_block
| error

attr_vals:
  attr_val
| attr_vals attr_val
| attr_vals ; attr_val

attr_val:
  attr_id SYM_EQ object_block

attr_id:
  V_ATTRIBUTE_IDENTIFIER
| V_ATTRIBUTE_IDENTIFIER error

object_block:
  complex_object_block
| primitive_object_block
| object_reference_block
| SYM_START_DBLOCK SYM_END_DBLOCK

complex_object_block:
  single_attr_object_block
| container_attr_object_block

container_attr_object_block:
  untyped_container_attr_object_block
| type_identifier untyped_container_attr_object_block

untyped_container_attr_object_block:
  container_attr_object_block_head keyed_objects SYM_END_DBLOCK

container_attr_object_block_head:
  SYM_START_DBLOCK

keyed_objects:
  keyed_object
| keyed_objects keyed_object

keyed_object:
  object_key SYM_EQ object_block
```


© 2013- The openEHR Foundation
Issue tracker: www.openehr.org/issues/browse/SPECPR Web: www.openEHR.org

```
object_key:
  [ primitive_value ]

single_attr_object_block:
  untyped_single_attr_object_block
| type_identifier untyped_single_attr_object_block

untyped_single_attr_object_block:
  single_attr_object_complex_head attr_vals SYM_END_DBLOCK
single_attr_object_complex_head:
  SYM_START_DBLOCK

primitive_object_block:
  untyped_primitive_object_block
| type_identifier untyped_primitive_object_block

untyped_primitive_object_block:
  SYM_START_DBLOCK primitive_object SYM_END_DBLOCK

primitive_object:
  primitive_value
| primitive_list_value
| primitive_interval_value
| term_code
| term_code_list_value

primitive_value:
  string_value
| integer_value
| real_value
| boolean_value
| character_value
| date_value
| time_value
| date_time_value
| duration_value
| uri_value

primitive_list_value:
  string_list_value
| integer_list_value
| real_list_value
| boolean_list_value
| character_list_value
| date_list_value
| time_list_value
| date_time_list_value
| duration_list_value

primitive_interval_value:
  integer_interval_value
| real_interval_value
| date_interval_value
| time_interval_value
| date_time_interval_value
| duration_interval_value
```


© 2013- The openEHR Foundation
Issue tracker: www.openehr.org/issues/browse/SPECPR Web: www.openEHR.org

```
type_identifier:
  ( V_TYPE_IDENTIFIER )
| ( V_GENERIC_TYPE_IDENTIFIER )
| V_TYPE_IDENTIFIER
| V_GENERIC_TYPE_IDENTIFIER

string_value:
  V_STRING

string_list_value:
  V_STRING , V_STRING
| string_list_value , V_STRING
| string_list_value , SYM_LIST_CONTINUE
| V_STRING , SYM_LIST_CONTINUE

integer_value:
  V_INTEGER
| + V_INTEGER
| - V_INTEGER

integer_list_value:
  integer_value , integer_value
| integer_list_value , integer_value
| integer_value , SYM_LIST_CONTINUE

integer_interval_value:
  SYM_INTERVAL_DELIM integer_value SYM_ELLIPSIS integer_value
SYM_INTERVAL_DELIM
| SYM_INTERVAL_DELIM SYM_GT integer_value SYM_ELLIPSIS integer_value
SYM_INTERVAL_DELIM
| SYM_INTERVAL_DELIM integer_value SYM_ELLIPSIS SYM_LT integer_value
SYM_INTERVAL_DELIM
| SYM_INTERVAL_DELIM SYM_GT integer_value SYM_ELLIPSIS SYM_LT integer_value
SYM_INTERVAL_DELIM
| SYM_INTERVAL_DELIM SYM_LT integer_value SYM_INTERVAL_DELIM
| SYM_INTERVAL_DELIM SYM_LE integer_value SYM_INTERVAL_DELIM
| SYM_INTERVAL_DELIM SYM_GT integer_value SYM_INTERVAL_DELIM
| SYM_INTERVAL_DELIM SYM_GE integer_value SYM_INTERVAL_DELIM
| SYM_INTERVAL_DELIM integer_value SYM_INTERVAL_DELIM

real_value:
  V_REAL
| + V_REAL
| - V_REAL

real_list_value:
  real_value , real_value
| real_list_value , real_value
| real_value , SYM_LIST_CONTINUE

real_interval_value:
  SYM_INTERVAL_DELIM real_value SYM_ELLIPSIS real_value SYM_INTERVAL_DELIM
| SYM_INTERVAL_DELIM SYM_GT real_value SYM_ELLIPSIS real_value
SYM_INTERVAL_DELIM
| SYM_INTERVAL_DELIM real_value SYM_ELLIPSIS SYM_LT real_value
SYM_INTERVAL_DELIM
| SYM_INTERVAL_DELIM SYM_GT real_value SYM_ELLIPSIS SYM_LT real_value
SYM_INTERVAL_DELIM
```


© 2013- The openEHR Foundation
Issue tracker: www.openehr.org/issues/browse/SPECPR Web: www.openEHR.org

```
| SYM_INTERVAL_DELIM SYM_LT real_value SYM_INTERVAL_DELIM
| SYM_INTERVAL_DELIM SYM_LE real_value SYM_INTERVAL_DELIM
| SYM_INTERVAL_DELIM SYM_GT real_value SYM_INTERVAL_DELIM
| SYM_INTERVAL_DELIM SYM_GE real_value SYM_INTERVAL_DELIM
| SYM_INTERVAL_DELIM real_value SYM_INTERVAL_DELIM

boolean_value:
  SYM_TRUE
| SYM_FALSE

boolean_list_value:
  boolean_value , boolean_value
| boolean_list_value , boolean_value
| boolean_value , SYM_LIST_CONTINUE

character_value:
  V_CHARACTER

character_list_value:
  character_value , character_value
| character_list_value , character_value
| character_value , SYM_LIST_CONTINUE

date_value:
  V_ISO8601_EXTENDED_DATE

date_list_value:
  date_value , date_value
| date_list_value , date_value
| date_value , SYM_LIST_CONTINUE

date_interval_value:
  SYM_INTERVAL_DELIM date_value SYM_ELLIPSIS date_value SYM_INTERVAL_DELIM
| SYM_INTERVAL_DELIM SYM_GT date_value SYM_ELLIPSIS date_value
SYM_INTERVAL_DELIM
| SYM_INTERVAL_DELIM date_value SYM_ELLIPSIS SYM_LT date_value
SYM_INTERVAL_DELIM
| SYM_INTERVAL_DELIM SYM_GT date_value SYM_ELLIPSIS SYM_LT date_value
SYM_INTERVAL_DELIM
| SYM_INTERVAL_DELIM SYM_LT date_value SYM_INTERVAL_DELIM
| SYM_INTERVAL_DELIM SYM_LE date_value SYM_INTERVAL_DELIM
| SYM_INTERVAL_DELIM SYM_GT date_value SYM_INTERVAL_DELIM
| SYM_INTERVAL_DELIM SYM_GE date_value SYM_INTERVAL_DELIM
| SYM_INTERVAL_DELIM date_value SYM_INTERVAL_DELIM

time_value:
  V_ISO8601_EXTENDED_TIME

time_list_value:
  time_value , time_value
| time_list_value , time_value
| time_value , SYM_LIST_CONTINUE

time_interval_value:
  SYM_INTERVAL_DELIM time_value SYM_ELLIPSIS time_value SYM_INTERVAL_DELIM
| SYM_INTERVAL_DELIM SYM_GT time_value SYM_ELLIPSIS time_value
SYM_INTERVAL_DELIM
| SYM_INTERVAL_DELIM time_value SYM_ELLIPSIS SYM_LT time_value
SYM_INTERVAL_DELIM
```


© 2013- The openEHR Foundation
Issue tracker: www.openehr.org/issues/browse/SPECPR Web: www.openEHR.org

```
| SYM_INTERVAL_DELIM SYM_GT time_value SYM_ELLIPSIS SYM_LT time_value
SYM_INTERVAL_DELIM
| SYM_INTERVAL_DELIM SYM_LT time_value SYM_INTERVAL_DELIM
| SYM_INTERVAL_DELIM SYM_LE time_value SYM_INTERVAL_DELIM
| SYM_INTERVAL_DELIM SYM_GT time_value SYM_INTERVAL_DELIM
| SYM_INTERVAL_DELIM SYM_GE time_value SYM_INTERVAL_DELIM
| SYM_INTERVAL_DELIM time_value SYM_INTERVAL_DELIM
```

date_time_value:

```
  V_ISO8601_EXTENDED_DATE_TIME
```

date_time_list_value:

```
  date_time_value , date_time_value
| date_time_list_value , date_time_value
| date_time_value , SYM_LIST_CONTINUE
```

date_time_interval_value:

```
  SYM_INTERVAL_DELIM date_time_value SYM_ELLIPSIS date_time_value
SYM_INTERVAL_DELIM
| SYM_INTERVAL_DELIM SYM_GT date_time_value SYM_ELLIPSIS date_time_value
SYM_INTERVAL_DELIM
| SYM_INTERVAL_DELIM date_time_value SYM_ELLIPSIS SYM_LT date_time_value
SYM_INTERVAL_DELIM
| SYM_INTERVAL_DELIM SYM_GT date_time_value SYM_ELLIPSIS SYM_LT
date_time_value SYM_INTERVAL_DELIM
| SYM_INTERVAL_DELIM SYM_LT date_time_value SYM_INTERVAL_DELIM
| SYM_INTERVAL_DELIM SYM_LE date_time_value SYM_INTERVAL_DELIM
| SYM_INTERVAL_DELIM SYM_GT date_time_value SYM_INTERVAL_DELIM
| SYM_INTERVAL_DELIM SYM_GE date_time_value SYM_INTERVAL_DELIM
| SYM_INTERVAL_DELIM date_time_value SYM_INTERVAL_DELIM
```

duration_value:

```
  V_ISO8601_DURATION
```

duration_list_value:

```
  duration_value , duration_value
| duration_list_value , duration_value
| duration_value , SYM_LIST_CONTINUE
```

duration_interval_value:

```
  SYM_INTERVAL_DELIM duration_value SYM_ELLIPSIS duration_value
SYM_INTERVAL_DELIM
| SYM_INTERVAL_DELIM SYM_GT duration_value SYM_ELLIPSIS duration_value
SYM_INTERVAL_DELIM
| SYM_INTERVAL_DELIM duration_value SYM_ELLIPSIS SYM_LT duration_value
SYM_INTERVAL_DELIM
| SYM_INTERVAL_DELIM SYM_GT duration_value SYM_ELLIPSIS SYM_LT
duration_value SYM_INTERVAL_DELIM
| SYM_INTERVAL_DELIM SYM_LT duration_value SYM_INTERVAL_DELIM
| SYM_INTERVAL_DELIM SYM_LE duration_value SYM_INTERVAL_DELIM
| SYM_INTERVAL_DELIM SYM_GT duration_value SYM_INTERVAL_DELIM
| SYM_INTERVAL_DELIM SYM_GE duration_value SYM_INTERVAL_DELIM
| SYM_INTERVAL_DELIM duration_value SYM_INTERVAL_DELIM
```

term_code:

```
  V_QUALIFIED_TERM_CODE_REF
| ERR_V_QUALIFIED_TERM_CODE_REF
```

term_code_list_value:

```
  term_code , term_code
```


© 2013- The openEHR Foundation
Issue tracker: www.openehr.org/issues/browse/SPECPR Web: www.openEHR.org

```
| term_code_list_value , term_code
| term_code , SYM_LIST_CONTINUE

uri_value:
  V_URI

object_reference_block:
  SYM_START_DBLOCK absolute_path_object_value SYM_END_DBLOCK

absolute_path_object_value:
  absolute_path
| absolute_path_list

absolute_path_list:
  absolute_path , absolute_path
| absolute_path_list , absolute_path
| absolute_path , SYM_LIST_CONTINUE

absolute_path:
  /
| / relative_path
| absolute_path / relative_path

relative_path:
  path_segment
| relative_path / path_segment

path_segment:
  V_ATTRIBUTE_IDENTIFIER [ V_STRING ]
| V_ATTRIBUTE_IDENTIFIER
```

B.1.2 Symbols

The following lexical analyser specification was extracted the .l file used in the Eiffel ODIN implementation:

```
----------/* definitions */ ----------------------------------------------
ALPHANUM [a-zA-Z0-9]
IDCHAR [a-zA-Z0-9_]
NAMECHAR [a-zA-Z0-9._\-]
NAMECHAR_SPACE [a-zA-Z0-9._\- ]
NAMECHAR_PAREN [a-zA-Z0-9._\-()]

UTF8CHAR (([\xC2-\xDF][\x80-\xBF])|(\xE0[\xA0-\xBF][\x80-\xBF])|([\xE1-
\xEF][\x80-\xBF][\x80-\xBF])|(\xF0[\x90-\xBF][\x80-\xBF][\x80-\xBF])|([\xF1-
\xF7][\x80-\xBF][\x80-\xBF][\x80-\xBF]))

----------/** Separators **/-----------------------------------------------

[ \t\r]+           -- Ignore separators
\n+                -- (increment line count)

----------/** comments **/-----------------------------------------------

"--".*             -- Ignore comments
"--".*\n[ \t\r]*   -- (increment line count)
```


© 2013- The openEHR Foundation
Issue tracker: www.openehr.org/issues/browse/SPECPR Web: www.openEHR.org

```
----------/* symbols */ ----------------------------------------------------
"-"         -- -> Minus_code
"+"         -- -> Plus_code
"*"         -- -> Star_code
"/"         -- -> Slash_code
"^"         -- -> Caret_code
"."         -- -> Dot_code
";"         -- -> Semicolon_code
","         -- -> Comma_code
":"         -- -> Colon_code
"!"         -- -> Exclamation_code
"("         -- -> Left_parenthesis_code
")"         -- -> Right_parenthesis_code
"$"         -- -> Dollar_code
"??"        -- -> SYM_DT_UNKNOWN
"?"         -- -> Question_mark_code

"|"         -- -> SYM_INTERVAL_DELIM

"["         -- -> Left_bracket_code
"]"         -- -> Right_bracket_code

"="         -- -> SYM_EQ

">="        -- -> SYM_GE
"<="        -- -> SYM_LE

"<"         -- -> SYM_LT or SYM_START_DBLOCK
">"         -- -> SYM_GT or SYM_END_DBLOCK

".."        -- -> SYM_ELLIPSIS
"..."       -- -> SYM_LIST_CONTINUE

----------/* keywords */ ------------------------------------------------

[Tt][Rr][Uu][Ee]        -- -> SYM_TRUE

[Ff][Aa][Ll][Ss][Ee]    -- -> SYM_FALSE

[Ii][Nn][Ff][Ii][Nn][Ii][Tt][Yy]      -- -> SYM_INFINITY

----------/* V_URI */ ---------------------------------------------------
[a-z]+:\/\/[^<>|\\{}^~"\[\] ]*

----------/* V_QUALIFIED_TERM_CODE_REF form [ICD10AM(1998)::F23] */ -----
\[{NAMECHAR_PAREN}+::{NAMECHAR}+\]

----------/* ERR_V_QUALIFIED_TERM_CODE_REF */ -----
\[{NAMECHAR_PAREN}+::{NAMECHAR_SPACE}+\]

----------/* V_LOCAL_TERM_CODE_REF form [local::at0004] */ -----
\[a[ct][0-9.]+\]

----------/* ERR_V_LOCAL_TERM_CODE_REF */ -----
\[{ALPHANUM}[^\]]+\]

--/* V_ISO8601_EXTENDED_DATE_TIME YYYY-MM-DDThh:mm:ss[,sss][Z|+/-nnnn] */ --
```


© 2013- The openEHR Foundation
Issue tracker: www.openehr.org/issues/browse/SPECPR Web: www.openEHR.org

```
[0-9]{4}-[0-1][0-9]-[0-3][0-9]T[0-2][0-9]:[0-6][0-9]:[0-6][0-9](,[0-9]+)?(Z|[+-][0-9]{4})? |
[0-9]{4}-[0-1][0-9]-[0-3][0-9]T[0-2][0-9]:[0-6][0-9](Z|[+-][0-9]{4})? |
[0-9]{4}-[0-1][0-9]-[0-3][0-9]T[0-2][0-9](Z|[+-][0-9]{4})?

----------/* V_ISO8601_EXTENDED_TIME hh:mm:ss[,sss][Z|+/-nnnn] */ --------
[0-2][0-9]:[0-6][0-9]:[0-6][0-9](,[0-9]+)?(Z|[+-][0-9]{4})? |
[0-2][0-9]:[0-6][0-9](Z|[+-][0-9]{4})?

----------/* V_ISO8601_EXTENDED_DATE YYYY-MM-DD */ ------------------------
[0-9]{4}-[0-1][0-9]-[0-3][0-9] |
[0-9]{4}-[0-1][0-9]

----------/* V_ISO8601_DURATION PnYnMnWnDTnnHnnMnn.nnnS */ -------------
-- here we allow a deviation from the standard to allow weeks to be
-- mixed in with the rest since this commonly occurs in medicine

P([0-9]+[yY])?([0-9]+[mM])?([0-9]+[wW])?([0-9]+[dD])?T([0-9]+[hH])?([0-9]+[mM])?([0-9]+(\.[0-
9]+)?[sS])? |
P([0-9]+[yY])?([0-9]+[mM])?([0-9]+[wW])?([0-9]+[dD])?

----------/* V_TYPE_IDENTIFIER */ ---------------------------------------
[A-Z]{IDCHAR}*

----------/* V_GENERIC_TYPE_IDENTIFIER */ -------------------------------
[A-Z]{IDCHAR}*<[a-zA-Z0-9,_<>]+>

----------/* V_ATTRIBUTE_IDENTIFIER */ ----------------------------------
[_a-z]{IDCHAR}*

----------/* CADL Blocks */ ----------------------------------------------
\{[^{}]*                    -- beginning of CADL block
<IN_CADL_BLOCK>\{[^{}]*     -- got an open brace
<IN_CADL_BLOCK>[^{}]*\}     -- got a close brace

----------/* V_INTEGER */ -----------------------------------------------
[0-9]+ |
[0-9]+[eE][+-]?[0-9]+

----------/* V_REAL */ --------------------------------------------------
[0-9]+\.[0-9]+|
[0-9]+\.[0-9]+[eE][+-]?[0-9]+

----------/* V_STRING */ ------------------------------------------------
\"[^\\\n"]*\"

\"[^\\\n"]*{                -- beginning of a multi-line string
<IN_STR> {
    \\\\                    -- match escaped backslash, i.e. \\ -> \
    \\\"                    -- match escaped double quote, i.e. \" -> "
    {UTF8CHAR}+             -- match UTF8 chars
    [^\\\n"]+               -- match any other characters
    \\\n[ \t\r]*            -- match LF in line
    [^\\\n"]*\"             -- match final end of string

    .|\n     |
    <<EOF>>                 -- unclosed String -> ERR_STRING
}
```


© 2013- The openEHR Foundation
Issue tracker: www.openehr.org/issues/browse/SPECPR Web: www.openEHR.org

```
----------/* V_CHARACTER */ ------------------------------------------
\'[^\\\n']\'     -- normal character in 0-127
\'\\n\'          -- \n
\'\\r\'          -- \r
\'\\t\'          -- \t
\'\\'\'          -- \'
\'\\\'           -- \\
\'{UTF8CHAR}\'   -- UTF8 char
\'.{1,2}   |
\'\\[0-9]+(\/)? -- invalid character -> ERR_CHARACTER
```


© 2013- The openEHR Foundation
Issue tracker: www.openehr.org/issues/browse/SPECPR Web: www.openEHR.org

© 2013- The openEHR Foundation
Issue tracker: www.openehr.org/issues/browse/SPECPR Web: www.openEHR.org

END OF DOCUMENT

© 2013- The openEHR Foundation
Issue tracker: www.openehr.org/issues/browse/SPECPR Web: www.openEHR.org