*open*EHR Reference Model Java ITS

Revision: 1.4

Pages: 18

Author: Rong Chen

rong@acode.se

ACODE HB, Sweden

http://www.acode.se

Revision History

Rev.	*Details*	*Who*	*Completed*
1.4	Incorporated T Beale's comments	R Chen	01/12/06
1.3	Some minor updates	R Chen	12/06/05
1.2	Added Demographic and EHR Extract information model	R Chen	03/13/05
1.1	Switched target JDK from 1.4 to 5.0	R Chen	10/15/04
1.0.2	“DV_” prefix kept and renamed to “Dv”	R Chen	09/07/04
1.0.1	Added “Implementations from Java API”, separated “immutability from value object” and some minor corrections	R Chen	08/27/04
1.0.0	Initial writing	R Chen	08/11/04

Acknowledgments

Thank Thomas Beale and others from *open*EHR for the original work, and Thomas Beale particularly for valuable discussions and the initial invitation to write this document.

Thank Göran Pestana, my partner at ACODE HB Sweden for the initial review and support.

1 Introduction

1.1 Purpose

This document provides recommendations and guidelines for Java implementation of the *open*EHR Reference Model, which at the time of writing consists of the Reference Information Model and the Archetype Model. The Reference Information Model includes Data Types, Demographic, Common, Support, Data Structures, EHR and EHR Extract model.

1.2 Related Documents

- The *open*EHR Data Types Information Model
- The *open*EHR Common Information Model
- The *open*EHR Demographic Information Model
- The *open*EHR Support Information Model
- The *open*EHR Data Structures Information Model
- The *open*EHR EHR Information Model
- The *open*EHR EHR Extract Information Model
- The *open*EHR Archetype Object Model
- The *open*EHR Archetype Definition Language
- The *open*EHR Archetype Profile

2 Background

2.1 Scope

This document is based on the work on a Java implementation of the *open*EHR Reference Model. The code has been released by ACODE under open source license and later adopted by the *open*EHR foundation as the reference Java implementation.

The recommendations presented here only apply to perspective Java implementation of the *open*EHR Reference Model. No presumption about usage of the reference model, e.g. used within desktop application or server-side component has been made. The idea is that the reference model implementation should be generic enough to be used in different application context. There is no design choice made for specific persistence or presentation solution either. Since in an archetype-based system, the reference model probably should not have any dependency on either the presentation layer or the persistence layer.

2.2 Guideline Criteria

The *open*EHR Reference Model is designed to be programming language independent. It has been developed according to the UM semantics, and implemented thus validated in Eiffel programming language.
There are a number of things one needs to bear in mind when implements the *open*EHR Reference Model in Java. Some of them are cosmetic issues, like naming conventions, some of them are more profound, for example, Design By Contract in Eiffel, which is heavily used to reinforce the reference model and but not supported natively in Java. In general, the goal is to implement the *open*EHR model in Java as faithful to the original specifications as possible and at same time keep the Java look-and-feel.

1. Java implementation must be able to present the information that can be presented by the original class model in the specifications.
2. It should look similar to the original model in terms of class names, attribute names and method signatures, so that mapping between the Java implementation and the original model can be easily made.
3. Java implementation should look like Java. That is it should follow Java standards, naming conventions and idioms, make use of both built-in and well-known APIs from third parties instead of re-inventing the wheel.

2.3 Java Platform

The Java platform targeted here is Java 2 Platform, Standard Edition, v5.0 mainly because of newly added support of generic types.
The Enterprise Edition is not required to implement the reference model, but it will be very useful in implementing the Service Model.
The 'assert' keyword added since J2SE v 1.4 could be useful to implement DBC like pre-conditions (see more in DBC section).

3. Assumed Types

The list of assumed types are quoted from *open*EHR Support Information Model document (support_im-1_1.pdf). The mapping to Java types[10] is listed below:

Assumed Type	***Java Type***	***Comment***
Any	Object	super class of all Java types
Boolean	boolean, Boolean	primitive type or class
Character	char, Character	primitive type or class
Integer	int, Integer	primitive type or class
Integer_64	long, Long	primitive type or class
Real	float, Float	primitive type or class
Double	double, Double	primitive type or class
String	String	Unicode is natively supported
Container	java.util.Collection	In Java, Container is

Assumed Type	***Java Type***	***Comment***
		not parent of arrays
Array<T>	T[]	supported as first-class objects
List<T>	java.util.List	subclass of Collection
Set<T>	java.util.Set	subclass of Collection
Bag<T>	**org.apache.commons.collections.bag**	from Apache Software Foundation
Interval	not supported	See Appendix.B for sample implementation.

4. Naming Convention

In general, naming of packages, classes, methods and attributes should follow the Java Naming Convention[11]

- packages
  DATA_TYPES.BASIC -> datatypes.basic
  In reality, one would put a full domain name in the path to make it unique, something like se.acode.openehr.datatypes.basic
- classes
  e.g. DATA_VALUE -> DataValue
  For all the subclasses of DataValue in datatypes package, “DV_” prefix of the class name is kept but renamed to “Dv” to follow the Java naming convention.
- fields
  e.g. calendar_alignment -> calendarAlignment
- methods
  e.g. is_strictly_comparable_to() -> isStrictlyComparableTo()
- accessors
  For fields (attributes) that are declared by the specification, they should be implemented as private fields and public accessors(getters) should be provided to access them.

5. Generic Types

Generics of *open*EHR Reference Model is implemented directly by Java generics, which is introduced since Java 5.0.

6. Design By Contract

There are several Java implementations of Design By Contract, e.g. iContract, which could be used in Java environment. The recommendations here could be used without involvement of any Java implementation of DBC.

1. Pre-conditions
   Pre-conditions for public methods that are related to parameters should be implemented as parameter validation and IllegalArgumentException or its subclass should be thrown when pre-conditions can not be satisfied. For non-public methods, pre-conditions can be implemented as assertions using keyword 'assert', which is added into Java since Java1.4. It worths to mention that pre-conditions for public methods that belong to parameter validation should not be implemented as assertions because these contracts must be obeyed whether assertions are enabled or not. Also bad parameters to the public methods should result in runtime exception instead of assertion failure. See Appendix.B for example.
2. Post-conditions
   Unit testing can be used to verify the result after execution of methods. Compared with DBC post-conditions, unit testing tests the result of specific condition by supplying known test data, while DBC post-conditions are more general. Because of that, it is probably easier to write unit tests than post-conditions[3].
3. Invariants
   Invariants can be implemented as parameter validation in the constructors if the object is immutable. Since the internal state does not change during the lifetime of any immutable object.

Otherwise invariants can also be implemented as methods and put at strategic places.

7. Value Object

Most of the datatypes classes are essentially Value Objects[2], whose purpose is to pass values around. The equality of Value Object is not based on the identity, but based on the values of contained fields.
Therefore, override both equals() and hashCode() for all Value Objects. Use the values of all relevant fields in equals() and hashCode() and obey general contract of both methods.
See Appendix.B for example

8. Immutable Object

Most of classes in the reference model are good candidates of immutable objects. Immutable objects are easier to design and implement than mutable ones. They are also easier to use and much safer. To make a class immutable, follow the following rules[15]:

1. Make all fields private and final
2. Make the class final
3. Only provide accessors methods to fields. Do not provide any mutators (setters).
4. Make defensive copies in constructors and accessors when fields refer to mutable objects, e.g. instances of Collection and its subclasses. For collection instances, one could use java.uitl.Collections.unmodifiableXXX() methods to get unmodifiable view of the collection.
   See Appendix.B for example

9. Exceptions

Parameter validation in public methods and constructors should throw IllegalArgumentException or its subclass if the specified pre-conditions can not be satisfied. It is important to throw Exceptions

that subclass RuntimeException instead of checked Exceptions to indicate that the failure is unrecoverable.
See Appendix.B for example

10. Multiple Inheritance

Multiple inheritance is not directly supported in Java. Luckily, there are few classes in the *open*EHR reference model that inherit multiple super classes. When required, one of the parent classes can be chosen to be implemented as the super class in Java, the others should then be implemented as fields. The criterion for choosing the parent class is based on its substitutability. For example, all subtypes of DataValue should be substitutable for DataValue, so any class that inherits DataValue really has to inherit it. The class datatypes.quantity.Interval is the only class in the entire Data Types model that multiply inherits - it also inherits assumed type Interval - but this is for implementation - no substitutability is needed. So it should inherit DataValue and include an instance of assumed type Interval as a field.

11. Abstract attributes

Abstract attributes is not supported in Java. The solution is declare it as abstract method and return the required type, then the subclass will simply implement it and return more specialized types.

12. Operator Overloading

Operator overloading is not supported in Java, therefore it should be implemented as method in the class. e.g.
infix '<' implemented as compareTo() of interface Comparable
infix '+' implemented as add() method
infix '-' implemented as substract() method
prefix '-' implemented as negate() method

13. Index

Index values of array and List always starts with 0 in Java. For example, the Java implementation of ithItem() method in ItemList takes index starting from 0.

14. String Representation

All classes should override the method toString() to provide human readable information. When appropriate, toString() should also present all interesting information contained in the object. See Appendix.B for example.

14.1 XML representation

XML based string representation can be useful and it should probably be required to be implemented for all Data Types classes. The XML string should include all non-calculated values from the object so that it would be possible to reconstruct the object later by parsing the XML string. The format of the XML string needs to be standardized by the *open*EHR, but one possibility is to use the attribute name as element name thus generating and parsing XML string can be automated by using Java Reflection API.

15. Comparable

Comparable interface should be implemented if instance of the class has natural order, e.g. datatypes.quantity.Ordered. By implementing Comparable interface, one could take advantage of many generic algorithms and collection implementations provided by the Java platform[15].

16. Implementations from Java API

There are existing classes from Java platform already provide concrete implementation of classes defined by *open*EHR.

- java.util.Calendar for implementation of classes in datatypes.quantity.datetime package, more specifically DvDate, DvTime and DvDateTime are simply thin wrappers of java.util.Calender

- java.util.TimeZone for field timezone in class datatypes.quantity.datetime.WorldTime
- java.net.URI for implementation of class datatypes.uri.URI
- org.ietf.jgss.Oid (included in J2SE1.4) for implementation of support.identification.ISO_OID
- java.util.UUID (included in J2SE1.5) for implementation of class support.identification.UUID

17. External APIs

There are well known external open source Java APIs that can provide solid implementation or test framework.

1. JUnit, a unit testing framework for Java.
   http://junit.org
2. Jakarta commons-lang, provides highly reusable static utility methods, chiefly concerned with adding value to java.lang and other standard core classes.
   http://jakarta.apache.org/commons/lang/
3. Jakarta commons-collections, contains implementations, enhancements and utilities that complements the Java Collections Framework
   http://jakarta.apache.org/commons/collections/
4. JDOM, XML API for manipulate XML in Java way,
   http://www.jdom.org

Appendix

A. References

A.1 General

1. Gamma E, Helm R, Johnson R, Vlissides J. Design patterns of Reusable Object-oriented Software. Addison-Wesley 1995
2. Fowler M, Patterns of Enterprise Application Architecture Addison Wesley 2003
3. Weirich J, Design by Contract and Unit Testing, http://onestepback.org/index.cgi/Tech/Programming/DbcAndTesting.html

A.2 openEHR

4. Beale T *et al*, Design Principles for the EHR
5. Beale T *et al*, *open*EHR Data Types Information Model
6. Beale T *et al*, *open*EHR Support Information Model
7. Beale T *et al*, *open*EHR Common Information Model
8. Beale T *et al*, *open*EHR Data Structures Information Model
9. Beale T *et al*, *open*EHR EHR Information Model
10. Beale T *et al*, *open*EHR Archetype Object Model
11. Beale T *et al*, *open*EHR Archetype Definition Language

A.3 Java

12. Java Language Specification, 2nd http://java.sun.com/docs/books/jls/
13. Java Naming Convention http://java.sun.com/docs/codeconv/html/CodeConventions.doc8.html
14. Eckel B, Thinking In Java, Prentice Hall PTR 1998
15. Bloch J, Effective Java, Addison-Wesley 2001

B. Sample implementation of Interval

```
package se.acode.openehr.support.basic;

import org.apache.commons.lang.builder.EqualsBuilder;
import org.apache.commons.lang.builder.HashCodeBuilder;

/**
 * Interval of comparable items. Instances of this class
 * are immutable.
 *
 * @author Rong Chen
 * @version 1.0
 */
public final class Interval {

    /**
     * Constructs an Interval
     *
     * @param lower null if unbounded
     * @param upper null if unbounded
     * @throws IllegalArgumentException if lower > upper
     */
    public Interval(Comparable lower, Comparable upper) {
        if (lower != null && upper != null
                && upper.compareTo(lower) < 0) {
            throw new IllegalArgumentException("lower > upper");
        }
        this.lower = lower;
        this.upper = upper;
    }

    /**
     * Returns the lower boundary of this Interval
     *
     * @return null if unbounded
     */
    public Comparable getLower() {
        return lower;
    }

    /**
     * Returns the upper boundary of this Interval
     *
     * @return null if unbounded
     */
    public Comparable getUpper() {
        return upper;
    }

    /**
     * Returns true if lower boundary open
     *
     * @return true if has lower boundary
     */
    public boolean isLowerUnbounded() {
        return lower == null;
    }

    /**
     * Returns true if upper boundary open
     *
     * @return true if has upper boundary
     */
    public boolean isUpperUnbounded() {
        return upper == null;
    }

    /**
     * Returns true if (lower >= value and value <= upper)
     *
     * @param value to compare to
     * @return ture if given value is included in this Interval
     * @throws IllegalArgumentException if value is null
     */
    public boolean has(Comparable value) {
        if (value == null) {
            throw new IllegalArgumentException("null value");
        }

        return ( lower == null || value.compareTo(lower) >= 0 )
                && ( upper == null || value.compareTo(upper) <= 0 );
    }

    /**
     * Equals if two Intervals have same values for lower and
     * upper boundaries
     *
     * @param o the object to compare with
     * @return true if equals
     */
    public boolean equals(Object o) {
        if (this == o) return true;
        if (!( o instanceof Interval )) return false;

        final Interval interval = (Interval) o;

        return new EqualsBuilder()
                .append(upper, interval.upper)
                .append(lower, interval.lower)
                .isEquals();
    }

    /**
     * Return a hash code of this Interval
     *
     * @return hash code
     */
    public int hashCode() {
        return new HashCodeBuilder()
                .append(upper)
                .append(lower)
                .toHashCode();
    }

    /**
     * Return string representation of this Interval. The string
     * consists of both lower and upper boundary, if any of them
     * is not specified, "unbounded" is provided.
     *
     * @return string representation
     */
    public String toString() {
        StringBuffer buf = new StringBuffer("Interval [lower: ");
        buf.append(lower == null ? "unbounded" : lower);
        buf.append(", upper: ");
        buf.append(upper == null ? "unbounded" : upper);
        buf.append("]");
        return buf.toString();
    }
    /* fields */
    private final Comparable lower;
    private final Comparable upper;
}
```

C. Sample Code of unit testing of Interval

```
/**
 * IntervalTest
 *
 * @author Rong Chen
 * @version 1.0
 */
package se.acode.openehr.support.basic;

import junit.framework.TestCase;

public class IntervalTest extends TestCase {

    public IntervalTest(String test) {
        super(test);
    }

    /**
     * The fixture set up called before every test method.
     */
    protected void setUp() throws Exception {
    }

    /**
     * The fixture clean up called after every test method.
     */
    protected void tearDown() throws Exception {
    }

    public void testConstructor() throws Exception {
        try {
            new Interval(new Integer(10), new Integer(1));
            fail("should throw illegal argument exception");
        } catch (Exception ignored) {
        }
    }

    public void testHas() throws Exception {
        // array of { lower(0:unbounded), upper(0:unbounded),
        //              testValue, expected (1:true, 0:false) }
        int[][] data = {
            {1, 8, 2, 1},
            {1, 8, 1, 1},
            {1, 8, 8, 1},
            {1, 8, 0, 0},
            {1, 8, 9, 0},
            {0, 8, 4, 1},
            {0, 8, -1, 1},
            {0, 8, 9, 0},
            {1, 0, 4, 1},
            {1, 0, 1, 1},
            {1, 0, -1, 0}
        };
        for (int i = 0; i < data.length; i++) {
            Interval iv = new Interval(popInt(data[ i ][ 0 ]),
                    popInt(data[ i ][ 1 ]));
            boolean actual = iv.has(new Integer(data[ i ][ 2 ]));
            boolean expected = data[ i ][ 3 ] == 1;
            assertTrue("failed at " + testString(data[ i ]),
                    actual == expected);
        }
    }

    private Integer popInt(int value) {
        if (value == 0) {
            return null;
        }
        return new Integer(value);
    }

    private String testString(int[] row) {
        return "(" + row[ 0 ] + ", " + row[ 1 ] + ") has " +
                row[ 2 ]
                + ": " + ( row[ 3 ] == 1 );
    }

    public void testToString() throws Exception {
        int[][] data = {
            {10, 100}, {0, 100}, {-20, 0}, {0, 0}
        };
        String[] expected = {
            "Interval [lower: 10, upper: 100]",
            "Interval [lower: unbounded, upper: 100]",
            "Interval [lower: -20, upper: unbounded]",
            "Interval [lower: unbounded, upper: unbounded]"
        };
        for (int i = 0; i < data.length; i++) {
            assertEquals(expected[ i ] + " expected",
                    expected[ i ],
                    new Interval(popInt(data[ i ][ 0 ]),
                            popInt(data[ i ][ 1 ])).toString());
        }
    }

    public void testEquals() throws Exception {
        Interval interval = new Interval(new Integer(-1), new
Integer(10));
        Interval interval2 = new Interval(new Integer(-1), new
Integer(10));
        assertEquals(interval, interval2);

        // not equals expected
        int[][] data = {
            {-1, 9}, {2, 10}, {0, 10}, {-1, 0}, {0, 0}
        };

        for(int i = 0; i < data.length; i++) {
            interval2 = new Interval(popInt(data[i][0]),
                    popInt(data[i][1]));
            assertFalse(interval2.toString(),
interval.equals(interval2));
            assertFalse(interval2.toString(),
interval2.equals(interval));
        }
    }
}
```