Java SIG: White Paper Outline

Introduction

The Java SIG is proposed as an HL7 special interest group. Its purpose is to foster and coordinate the development of a Java language API to the HL7 RIM and its many artifacts. The goal of the API is to promote the development of different interoperable healthcare applications that not only conform to the HL7 standard but also take advantage of the same platform regardless of computing equipment used in the application.

When interested HL7 members sought the approval of the HL7 Technical Steering Committee and the HL7 Board of Directors for creation of the Java SIG, they characterized the mission and charter of the SIG as follows:

- [Mission] This group will define application programming interfaces (APIs) to HL7 version 3 artifacts for the Java platform. While the group's focus will be the definition of APIs, it will also promote the development of implementations to validate the API, to serve as reference implementations, and as tools. By providing these APIs and promoting implementations, this group will hasten the adoption of HL7 version 3 by healthcare application providers, many of whom want to develop for the Java platform.

  By doing this work for the Java platform, the group will also incidentally provide a model for HL7 implementations that assure interoperability in other popular programming languages, further encouraging development of HL7 Version 3-compliant applications.

- [Charter] The group will consider many use cases and applications for the objects it defines, including but not limited to messaging (interoperable with the HL7 v3 XML ITS), documents, and "component object" support of application roles. Applicability will be as Java-based messaging / interface components as well as intelligent interfaces to HL7 data objects (i.e., RIM and R-MIM objects) for use with other ITS platforms, such as applications that are primarily XML-based.

  The scope of HL7 artifacts addressed in this Java API effort includes data types, message definitions, the CDA, the reference information model, trigger events, interactions, and application roles.

The remainder of this white paper is based on a monograph produced by Gunther Schadow, MD, PhD, of the Regenstrief Institute at the Indiana University School of Medicine. It describes particular use cases in which a Java API for HL7 would be valuable, and it proposes different approaches for harvesting Javatized versions of HL7 artifacts. It offers a starting point for design and planning discussions for the Java SIG.

Use Cases for a Java API to HL7

After having done some mapping of HL7 stuff to Java, we found that we needed to go back to the roots and look at the use cases first to avoid bathing in what's fun to do but

overlooking what's really needed and useful. So, I am thinking up some use cases where I think the Java API and implementation would be most needed.

1. XML processing of messages for transformation and validation and more
2. Developing an all-new Java-based health care application
3. Adding HL7 to an existing Java-based health care application
4. Adding HL7 to a non-Java health care application
5. Writing a Java based messaging processor
6. Translating between HL7 v2.x and v3

XML processing of messages for transformation and validation and more

I put this use case first because it is perhaps the one most urgently needed piece. Many if not most people who got recently interested in HL7 v3 come from the XML side. So, their main interest is handling XML messages and documents.

These people, as well as HL7 designers themselves, will find or have already found that XML alone doesn't give them all they need to handle HL7 information well. This is certainly true for well-formed XML and XML with DTDs, but it even applies for XML with XML Schema.

Examples:

- Most restrictions on coded data types use the sub-class (aka "ISA") relation instead of exact equality. This is done to provide for extensibility but it is a serious issue that anyone will run into at some time (at the latest when encountering a case where the code sent is not equal but only a specialization to the expected concept).
- Units of measure have always been a problem. If a standard code is used, conversions between most customary and metric units or different metric conventions can be done seamlessly. But if the interface person has to do this with just the concepts of numbers and strings, it is too hard for most to do it right. This will lead to many problems, especially because HL7 does not constrain the EXACT unit. For example, it is unreasonable for HL7 to prefer 1 m for length over 1 cm, and, it is historically a hard problem to convince anybody in the US not to use 1 inch as the preferred unit of length.
- The whole area about time and especially timing (repeated schedules) is quite hard. Leaving implementers with only a specification and the tools of strings and numbers leaves an overwhelming problem to solve for the implementers. They will often have to resort to kludges that will be unsafe and hardly interoperable.
- Some points of the HL7 message design cannot be readily expressed in XML-Schema, or even if they can the XML Schema is fairly complicated (e.g., using much extensions and restrictions which requires to issue and consider xsi:type assertions.)
- Many of HL7's constraints on the message content (elements and vocabulary, etc.) is not expressible through XML Schema. Checking these constraints is sometimes left to the interface/application program. This portion of validation tends to increase. While

validation isn't always required on a receiving interface, it is safer to validate. Also, for conformance testing and general prototyping and validation of interfaces as well as the HL7 specification itself, both developers and the HL7 organization have a stake in a validation environment that can validate against all of the HL7 specification at the appropriate level of abstraction.

There are multiple ways to manipulate XML instances: serial event-driven model (SAX), random access model (DOM), used in programming languages like Java, or some XML scripting environment that may connect to Java. It appears as if XSLT has really gotten off the ground in the industry and is a very powerful tool for XML manipulation, and for that reason I would focus on XSLT as the primary environment for this use case.

XSLT can be extended by a half-way standard method. A very common reasonably fast, good, and free XSLT processor is Saxon (written by the co-chair of the W3C XSLT committee himself). Saxon is written in Java and makes including Java class libraries extremely easy. All one needs to do is declare an extension namespace prefix for a Java class, from which point on one can access all the classes' methods in XSLT/XPath expressions. For example, one could import the PhysicalQuantity (PQ) and ConceptDescriptor (CD) data types from the Java API and, say, a generic data value instance factory:

```
<xsl:transform version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:pq="java:org.hl7.datatypes.PhysicalQuantity"
  xmlns:cd="java:org.hl7.datatypes.ConceptDescriptor"
  xmlns:f="java:org.hl7.InstanceFactory"
  xmlns:rc="java:org.regenstrief.rmrs.dictionary.Concept">
```

First we will have to make a static instance of the instance factory:

```
  <xml:variable name="factory" select="f:get-factory()"/>
```

Saxon will then call the (static) Java method:

```
  org.hl7.InstanceFactory.getFactory();
```

and Saxon will from now on know the Java *object* returned from that method under the name $factory.

Let's say we have an XML template for an observation message, and the object is to copy the observation. Using an interface to our Regenstrief dictionary we could want to see if the value conforms to the type of observation and, if applicable, convert the unit to our local standard convention, and then output an HTML table row element for the information.

```
<xsl:template match="*[xsi:type='Observation']">
```

First we put the observation code and the value of the observation into Java objects using the factory:

```
 <xsl:variable name='observation-code'
        select="f:makeCD($factory,
            ./cd/nullFlavor,
            ./cd/codeSystem,
            ./cd/code);"/>
```

This is like the Java code:

```
factory.makeCD(nullFlavor, codeSystem, code);
```

with the arguments being basically Strings at this point.

The same way we grab the value, assuming that it is a physical quantity. (In reality, of course, either the XSLT script or the factory would need to provide for alternatives.)

```
<xsl:variable name='observation-value'
    select="f:makePQ($factory,
                     ./cd/nullFlavor,
                     ./cd/value,
                     ./cd/unit);"/>
```

Now I have two XSLT variables $observation-code and $observation-value, bound to Java objects. Next I will get a descriptor object for the code out of the Regenstrief dictionary. This will tell me the preferred unit:

```
<xsl:variable name='rmrs-parameter'
              select="rc:new($observation-code)"/>
```

This in turn calls the Java constructor method and magically looks up the HL7 concept descriptor in a database to return the Regenstrief Concept object. If either the concept is not found or the expected kind of quantity does not conform, report an error, else show the preferred name value and unit.

```
 <tr>
  <xsl:choose>
   <xsl:when test="$rmrs-parameter">

     <xsl:variable name='rmrs-unit'
       select="rc:get-preferred-unit($rmrs-parameter)"/>

     <xsl:choose>
       <xsl:when test="pq:compares($rmrs-unit, $observation-value)">
         <td>
     <xsl:value-of select="rc:get-preferred-name($rmrs-parameter)"/>
           [<xsl:value-of select="$rmrs-unit"/>]
         </td>
         <td>
           <xsl:value-of select="pq:times($observation-value,
                                 pq:invert($rmrs-unit))"/>
         </td>
       </xsl:when>
       <xsl:otherwise>
         <td colspan=2>ERROR: incomparable units</td>
       </xsl:otherwise>
     </xsl:choose>
   </xsl:when>
   <xsl:otherwise>
     <td colspan=2>ERROR: unknown observation code</td>
   </xsl:otherwise>
 </xsl:choose>
</tr>
```

Conclusion: even given the minimal API for data types, powerful operations can be performed in XSLT that would otherwise be impossible. Using Java in XSLT with Saxon may also call for some more convenient methods and wrappers to make things even easier. However, the important point is that with just the standard XML data types string

and number, this would have been nearly impossible.

Developing an all-new Java-based health care application

Let's say we want to write an order entry system. Some of the critical components one would need is classes for orders, observation, signing physicians, roles, etc. The HL7 RIM provides a nice view on these objects. Since these objects are also the heart of the application program, it is quite likely that the application programmer would either extend if not rewrite the HL7 RIM classes that would come with a Java class library. Even if the programmer makes these classes all new, he can still make them implement the HL7 RIM class interface.

Clearly there is a need to bind these objects to a database. Some standard Object Relational mapper would be helpful, or perhaps just a transparent persistence environment where Java objects are paged in and out of main memory to secondary storage would suffice.

To send a created order via HL7, given that all objects implement the RIM (or R-MIM) interfaces, the programmer could just use something like

```
messenger.createNewOrder(myOrder);
```

and the object graph would be cast into HL7 v3 conformant XML and sent by means of however the "messenger" object was configured.

But in this use case we shouldn't focus on messaging alone. The point here is that the application programmer can work on the exact same high level of abstraction that the HL7 standard takes on.

However, clearly it is important that this standard API be highly flexible, because there will have to be a lot of functionality that the application will need to add, and there are a lot of architecture, design, and performance issues that the standard API should be ignorant about, but should enable.

Adding HL7 to an existing Java-based health care application

Here the application may use very different conceptualizations than HL7 uses. In this case the HL7 influence will probably be limited to the interface. An interface routine would then instantiate very simple data holder objects according to the RIM / R-MIM / HMD and then would copy data from the application data structures into the HL7 data structures just like filling out a form.

One could imagine this to be done as follows:

```
HL7Message msg = HL7Message.makeTemplate("POLB_MT009876");
```

This would return a template as a minimal object graph for this message. It would be a mutable object, and the idea is to assign values to the data fields.

```
msg.observationOrder.setCode(myApp.theOrder.serviceCode);
msg.observationOrder.setEffectiveTime(myApp.theOrder.deadline);
...
```

One could also imagine that the templates could be pre-populated with data that is defined in some configuration file, such that the application program only needs to

program those value assignments that are actually unique to the current instance of an order.

One might go as far as to define data mapper configuration, such that all this copying and assigning and converting of data could be configured in some file. For example, one would have an HL7 XML message template in XML where one could use special XML extensions or value strings that would correspond to certain objects of the application—almost as if application objects were dumped to XML and then an XSLT transform were written to generate the HL7 message.

Adding HL7 to a non-Java health care application

If Java isn't the primary language of the application, one would probably again use an XSLT transformation approach rather than calling a JVM from, say, Visual Basic.

But if the application programming environment allows for binding in Java beans, one can again do much of the magic shown with XSLT above. For instance, one could have a beans interface to the Tcl/TK scripting language, or, certainly, a JSP application could use the HL7 beans.

Writing a Java based messaging processor

Not to overdo the XSLT too much, but one could try that in XSLT. But perhaps one would want a more comprehensive application or some GUI-based message-processing management. The two things to tackle by such an application would be, again, mapping information but also managing the messaging transactions.

For an HL7-based document management system the situation would be very similar.

Translating between HL7 v2.x and v3

This is like translating between application data and HL7. It could be done by XSLT as well.

Conclusion on Use Cases

I may have missed important other use cases, and I most certainly had a limited perspective on some or all use cases. I keep falling back to XSLT for example. Also I keep emphasizing the very basic stuff of HL7 data objects and glossing over message handling. We need to consider this further with the SIG.

Mapping the HL7 Artifacts to Java

Data Types

Data types are defined in the document entitled "Data Types, Unabridged Specification":

http://www.hl7.org/v3ballot/html/foundationdocuments/helpfiles/datatypes.htm

The XML rendition of these data types is defined in the document entitled "HL7 v3 Data Types Implementable Technology Specification for XML":

http://www.hl7.org/v3ballot/html/foundationdocuments/ITSXML/ITSXMLdt.htm

The latter document is to a goodly extent self-consistent, and some people prefer using

that as the only reference. That may work to an extent for XML, but for Java we need to go back to the roots.

Unfortunately though the first document as per the web site is in pretty bad shape; it's somewhat outdated as well as rendered such that the key stuff is pretty hard to read. But I am copying the samples that I will use.

The unabridged specification contains a formal data type definition and constraint language that is very close to common Java syntax. So it will be quite simple to transform that into Java. (At this very moment none of the content of these formal language expressions has been cast into XML, so the transformation may use some means other than XSLT.)

Here are a few example that show the transformation.

All data types are rooted in a single class DataValue or ANY for short whose interface is defined thus:

```
abstract type DataValue alias ANY {
          DataType  dataType;
          BL        nonNull;
          CS        nullFlavor;
          BL        isNull;
          BL        notApplicable;
          BL        unknown;
          BL        other;
          BL        equals(ANY x);
};
```

This translates directly into a Java interface:

```
interface ANY {
          DataType dataType();
          BL        nonNull();
          CS        nullFlavor();
          BL        isNull();
          BL        notApplicable();
          BL        unknown();
          BL        other();
          BL        equals(ANY x);
}
```

People felt strongly about this short name of data types (like "BL" for "Boolean") to be preferred. For Java, we could go either way, and I assume that the long names would be better. Just for my laziness I will stick to the short names here; once this process is automated, it's easy to change that.

The Boolean type is defined as

```
type Boolean alias BL extends ANY
    values(true, false)
{
          BL        and(BL x);
          BL        not;
literal   ST;
          BL        or(BL x);
          BL        eor(BL x);
          BL        implies(BL x);
};
```

This, again, translates easily into Java interfaces as

```
interface BL extends ANY {
          BL         and(BL x);
          BL         not();
          BL         or(BL x);
          BL         eor(BL x);
          BL         implies(BL x);
}
```

I am leaving out the notions of literals and anything that has to do with instantiation, manipulation, and rendering.

The values true and false would be singletons. I'm not sure right now how to do that in the purity of an interface without dragging constructors/factory object into the definition of the interface. Presumably I could use static methods true() and false() respectively, but I'm not sure right now if that is possible in Java. (I know I can do public static final fields, but don't see explicit mention of static methods in the JLS chapter 9.)

Here, as a sample, the HL7 v3 data types specify the semantics of its properties by text and some very concise formal invariants. For instance, to find out what "BL implies(BL x)" means you find the following:

> 2.1.1.4 Implication
>
> The logical implication is important to make invariant statements. An implication is a rule of the form IF condition THEN conclusion. Logically the implication is defined as the disjunction of the negated condition and the conclusion, meaning that when the condition is true the conclusion must be true to make the overall statement true.
>
> ```
> invariant(BL condition, conclusion) {
>  condition.implies(conclusion).equals(condition.not.or(conclusion));
> };
> ```
>
> The implication is not reversible and does not specify what is true when the condition is false (*ex falso quodlibet*).

So, Java implementations fall out of this in a pretty straightforward fashion:

```
class BLimpl implements BL ... {
  ...
  BL implies(BL conclusion) {
    return this.not().or(conclusion);
  }
}
```

Moving on, Binary data is defined as

```
protected type BinaryData alias BIN extends LIST<BL>;
```

So, it says that BinaryData is a sequence of booleans. Therefore, given the Java generics and the HL7 collections, that's easy to express:

```
interface BIN extends LIST<BL> {
}
```

Note: Realize that the notion of HL7 collections such as SET and LIST is *broader* than the Java Collection to accommodate continuous domains. In particular iterators for

singular elements are not promised on the interface. Also, HL7 collections can be immutable, and the interface would not contain any mutating functions at this level. (At another level, all necessary convenience methods would be available to build these objects.)

Next is the encapsulated data:

```
type EncapsulatedData alias ED extends BIN {
          CS        type;
          CS        charset;
          CS        language;
          CS        compression;
          TEL       reference
          BIN       integrityCheck;
          CS        integrityCheckAlgorithm;
          ED        thumbnail;
          BL        equals(ED x);
};
```

This is how it appears when converted to Java:

```
interface ED extends BIN {
          CS        type();
          CS        charset();
          CS        language();
          CS        compression();
          TEL       reference()
          BIN       integrityCheck();
          CS        integrityCheckAlgorithm();
          ED        thumbnail();
          BL        equals(ED x);
};
```

This is so easy. Just replace the header with "interface" and put parentheses after each member name that doesn't already have them. I continue with this just a little bit because it exposes some of the cool edges about the HL7 v3 data types. Next comes String, which is very much a "derived" data type; it is a restriction of ED.

```
type CharacterString alias ST restricts ED {
          INT       length;
          ST        head;
          ST        tail;
};
invariant(ST x) where x.nonNull {
  x.type.equals("text/plain");
  x.compression.notApplicable;
  x.reference.notApplicable;
  x.integrityCheck.notApplicable;
  x.integrityCheckAlgorithm.notApplicable;
  x.thumbnail.notApplicable;
}
```

So, what restriction means is that you have many of the properties limited to fixed or tightly derived values. We cannot express this in interfaces very well, unfortunately, but the implementation would do that.

```
interface ST extends ED {
          INT       length();
          ST        head();
          ST        tail();
```

```
};
```

The length, head, and tail methods inherited originally from LIST<BL> are overridden here to be based on the element of a character rather than a single bit.

One implementation could be an adapter class for java.lang.String and look like this:

```
class STimpl {
  String _value;

  INT length() { return INTfactory.make(_value.length()); }
  ST head() { return new STimpl(_value.substring(0,1)); }
  ST tail() { return new STimpl(_value.substring(1)); }

  CS type() { return CSimpl.textPlain; }
  CS compression() { return CSimpl.notApplicable; }
  TEL reference()  { return TELimpl.notApplicable; }
  BIN integrityCheck() { return BINimpl.notApplicable; }
  CS integrityCheckAlgorithm() { return CSimpl.notApplicable; )
  ED thumbnail() { return EDimpl.notApplicable; }
}
```

This shows how the Java side can reflect pretty much one-to-one what the HL7 specification says. However, it raises the obvious question of what the most useful interface is. It seems odd that an HL7 program would work with a special form of string adapter rather than directly with java.lang.String. At this point we need to note two things:

- It would all be easier if java.lang.String wasn't declared final.
- The HL7 data type specification tries to be as general, technology-independent and all-encompassing as possible, but it explicitly does not require all implementation specifications to be as general and all-encompassing as well.

So, it should be fine to use just java.lang.String for strings, except for the cases when one of the null values needed is defined in the ANY type or any other HL7-specific functionality. I hesitate to use pairs of data and "indicator" fields in the form that I have seen with some SQL APIs, but it is one possibility. On the other hand, encapsulating strings in an adapter is not that much of a problem either, because a true HL7 program will not have much business dealing with java.lang.String things. Usually we will deal with codes and other data types that are much more high level and have useful operations instead of requiring the user to do string operations.

Let's fast forward to the code data type then, because it is so important in medicine.

```
type ConceptDescriptor alias CD extends ANY {
          ST         code;
          ST         displayName;
          OID        codeSystem;
          ST         codeSystemName;
          ST         codeSystemVersion;
          ED         originalText;
          LIST<CR>   modifier;
          SET<CD>    translation;
          BL         equals(CD x);
          BL         implies(CD x);
  demotion           ED;
};
```

This is how it appears when converted to Java:

```
interface CD extends ANY {
          ST         code();
          ST         displayName();
          OID        codeSystem();
          ST         codeSystemName();
          ST         codeSystemVersion();
          ED         originalText();
          LIST<CR>   modifier();
          SET<CD>    translation();
          BL         equals(CD x);
          BL         implies(CD x);
};
```

(I am not dealing with the concept of implicit type conversions at this point.)

So, that was easy again. The power of this data type will be the operation "implies(CD x)" and its implementation will be involved. This is the most commonly used operation with this data type, more common than "equals." Implementations of the CD type will probably be very much aware of their specific coding system. So, an ICD9-CD will be implemented differently than a SNOMED-CD. There will be a number of coding systems that don't have to have a special implementation class. Some code systems are flat, and then "implies" is the same as "equals." One can also make an implementation of this that uses certain "terminology server" resources that might potentially bridge between different coding systems.

As an example of an important quantitative data type in medicine, I take the Physical Quantity:

```
type PhysicalQuantity alias PQ extends QTY {
          REAL       value;
          CS         unit;
          BL         equals(PQ x)
          BL         lessOrEqual(PQ x);
          BL         compares(PQ x);
          PQ         canonical;
  type    PQ         diff
          diff       minus(PQ x);
          PQ         plus(diff x);
          PQ         negated;
          PQ         times(REAL x);
          PQ         times(PQ x);
          PQ         inverted;
          PQ         power(INT x);
  literal ST;
  demotion           REAL;
};
```

And here is the Javatized version:

```
interface PQ extends QTY {
          REAL       value();
          CS         unit();
          BL         equals(PQ x)
          BL         lessOrEqual(PQ x);
          BL         compares(PQ x);
          PQ         canonical();
  interface diff extends PQ { }
          diff       minus(PQ x);
```

```
          PQ        plus(diff x);
          PQ        negated();
          PQ        times(REAL x);
          PQ        times(PQ x);
          PQ        inverted();
          PQ        power(INT x);
}
```

The use of the "diff" interface may not be doable this way in Java, but I'm sure there is a workaround. The PQ data type allows people to deal with dimensioned quantities without being concerned about the exact units. Unit conversions are automatic. I have implemented this twice before, and it's really cool.

Now let's look at some collections, first the set:

```
template<ANY T>
type Set<T> alias SET<T> extends ANY {
          BL        contains(T element);
          BL        isEmpty;
          BL        nonEmpty;
          BL        contains(SET<T> subset);
          INT       cardinality;
          SET<T>    union(SET<T> otherset);
          SET<T>    except(T element);
          SET<T>    except(SET T  otherset);
          SET<T>    intersection(SET<T> otherset);
  literal ST;
  promotion         SET<T>    (T x);
};
```

And here is the Javatized version:

```
interface SET<T implements ANY> extends ANY {
          BL        contains(T element);
          BL        isEmpty();
          BL        nonEmpty();
          BL        contains(SET<T> subset);
          INT       cardinality();
          SET<T>    union(SET<T> otherset);
          SET<T>    except(T element);
          SET<T>    except(SET T  otherset);
          SET<T>    intersection(SET<T> otherset);
};
```

This was extremely straightforward. Notice that there is no iterator on the general SET interface; this is because of a large and important class of continuous sets. For them we often have a special kind of set, the interval:

```
template<QTY T>

type Interval<T> alias IVL<T> extends SET<T> {
          T         low;
          BL        lowClosed;
          T         high;
          BL        highClosed;
          T.diff    width;
          T         center;
          IVL<T>    hull(IVL<T> x);
  literal ST;
  promotion         IVL<T>    (T x);
  demotion T;
};
```

And here is the Javatized version:

```
interface IVL<T implements QTY> extends SET<T> {
          T          low();
          BL         lowClosed();
          T          high();
          BL         highClosed();
          T.diff     width();
          T          center();
          IVL<T>     hull(IVL<T> x);
};
```

Cool! Thanks to the Java generics, it is really straightforward, and better than in C++, where the "implements QTY" constraint wasn't possible to check at compile time.

Another quick look at the mother of all sequences (including continuous ones):

```
template<ANY T>
type Sequence<T> alias LIST<T> extends ANY {
          T          head;
          LIST<T>    tail;
          BL         isEmpty;
          BL         nonEmpty;
          INT        length;
  literal ST;
  promotion          LIST<T>    (T x);
};
```

And here again is the Javatized version:

```
interface LIST<T implements ANY> extends ANY {
          T          head();
          LIST<T>    tail();
          BL         isEmpty();
          BL         nonEmpty();
          INT        length();
};
```

It works.

Finally we have the notion of "generic type extensions" that add certain optional features to other data types. For instance, history item (HXIT) has an interval of point in time (TS):

```
interface HXIT<T implements ANY> extends T {
          IVL<TS>    validTime;
};
```

I am anxious to *try*, if this construct is allowed in Java generics, i.e., that the generic interface extends its parameter type.

There is a lot more of thinking and cooking needed to strike the best balance between a correct and close reflection of the high level HL7 data types and some of the constraints of what's possible in Java and what will perform well. Data types in HL7 are not no-brainers and are designed to be useful on a high level—they are much more than just capsules to pack and unpack dumb register values.

RIM Classes

Now let's do some RIM classes. The core backbone is Entity, Role, Participation, and

Act:

```
interface Entity {
            CS       class_cd();
            CS       determiner_cd();
            SET<II>  id();
            CE       cd();
            SET<PQ>  qty();
            BAG<EN>  nm();
            ED       desc();
            CS       status_cd();
            IVL<TS>  existence_time();
            BAG<TEL> telecom();
            CE       risk_cd();
            CE       handling_cd();
}
```

This is all very straightforward, and as an interface to an object that is instantiated or constructed somewhere outside the picture, it works very well. But, of course, with these data objects we often want to think of them as beans:

```
interface Entity {
            CS       getClassCd();
            CS       getDeterminerCd();
            SET<II>  getId();
            CE       getCd();
            SET<PQ>  getQty();
            BAG<EN>  getNm();
            ED       getDesc();
            CS       getStatusCd();
            IVL<TS>  getExistenceTime();
            BAG<TEL> getTelecom();
            CE       getRiskCd();
            CE       getHandlingCd();
}
```

Adding in the setters is a simple lexical operation on this class definition, but I am not doing it in this case.

[I am raising the age-old question of whether the getFoo() form is preferable over the foo() form of the property. The getFoo() form will be handled by beans-aware utilities, and I do believe in the ability to use beans in things like JSP or scripting languages. On the other hand, using this in Java programs directly, the get before getFoo() becomes cumbersome. I guess it is not a problem with these RIM classes that are merely data holders. It is a bigger issue with the data types that are not just data holders and where most of the properties would never have set accessors but are rather "functions." It's a bit unfortunate how the getFoo() style forces one on the java level to make distinctions between properties with and without arguments.]

On with the other classes in bean form, and now I will add the associations to the game:

```
interface Entity {
  CS       getClassCd();
  CS       getDeterminerCd();
  SET<II>  getId();
  CE       getCd();
  SET<PQ>  getQty();
  BAG<EN>  getNm();
  ED       getDesc();
```

```
  CS       getStatusCd();
  IVL<TS>  getExistenceTime();
  BAG<TEL> getTelecom();
  CE       getRiskCd();
  CE       getHandlingCd();

  SET<Role> getPlayedRole();
  SET<Role> getScopedRole();
}

interface Role {
  CS       getClassCd()
  SET<II>  getId()
  CE       getCd()
  BL       getNegationInd()
  BAG<AD>  getAddr()
  BAG<TEL>getTelecom()
  CS       getStatusCd()
  IVL<TS>  getEffectiveTime()
  ED       getCertificateTxt()
  RTO      getQty()
  LIST<INT> getPositionNbr()

  Entity   getPlayer();
  Entity   getScoper();
  SET<Participation> getParticipation();
}
interface  Participation {
  CS       getTypeCd();
  CD       getFunctionCd();
  CS       getContextControlCd();
  INT      getSequenceNbr();
  ED       getNoteTxt();
  IVL<TS> getTime();
  CE       getModeCd();
  CE       getAwarenessCd();
  CS       getSignatureCd();
  ED       getSignatureTxt();

  Role     getRole();
  Act      getAct();
}
interface Act {
  CS       getClassCd();
  CS       getMoodCd();
  SET<II>  getId();
  CD       getCd();
  BL       getNegationInd();
  ED       getText();
  CS       getStatusCd();
  GTS      getEffectiveTime();
  GTS      getActivityTime();
  TS       getAvailabilityTime();
  SET<CE>  getPriorityCd();
  SET<CE>  getConfidentialityCd();
  IVL<INT> getRepeatNbr();
  BL       getInterruptibleInd();
  BL       getContext_lockInd();
  BL       getIndependentInd();
  SET<CE>  getReasonCd();
  CE       getLanguageCd();
```

```
  SET<Participation>   getParticipation();
  SET<ActRelationship> getSourcingRelationship();
  SET<ActRelationship> getTargetingRelationship();
}
interface ActRelationship {
  CS       getTypeCd();
  BL       getInversionInd();
  CS       getContextControlCd();
  INT      getSequenceNbr();
  INT      getPriorityMbr();
  PQ       getPauseQty();
  CS       getCheckpointCd();
  CS       getSplitCd();
  CS       getJoinCd();
  BL       getNegationInd();
  CS       getConjunctionCd();

  Act      getSource();
  Act      getTarget();
}
```

The key here will lie in the flexible definition and intelligent implementation of the association accessors. Consider the case where the data resides in a database of sorts, and you do not want to read entire object graphs from that database. (In most cases, the whole database will be somehow connected, and so loading the entire object graph means loading the entire database.) So, the approach I suggest for the implementation will involve weak references and more intelligent accessors that can use criteria to select only a subset of all elements of an association with cardinality > 1. For instance, when you want only act relationships of type "has component" (COMP) you should be able to say something like:

```
Act a;
SET<Act> componentActs
   = a.getSourcingRelationship(/* x where x.typeCd.implies(COMP) */).
```

The question will be how general we can make this kind of selection criterion. If all things were Java, one could use an

```
interface SelectionCriterion {
  BL test(ANY x);
}
```

Then the Act above could become

```
SET<Act> componentActs
   = a.getSourcingRelationship(new SelectionCriterion {
         test(ActRelationship x) {
             BL x.typeCd.implies(ActRelationshipType.COMP);
           }
      });
```

That's nice. However, if we assume that we have a relational database, one would want to push the selection criterion off to the SQL server, and in this case one might have to do something more kludgy. Particularly one would have to do some thinking about how to do the `typeCd.implies(ActRelationshipType.COMP)`operation, because that's far more than just `... WHERE type_cd = 'COMP'` and SQL implementations are notoriously bad (incompatible at best) at abstract data types. The key will be to define the

API such that

- Intelligent implementations are possible and can perform well.
- Clever tricks can be done (e.g., the "implies" operation can be emulated with two integers:

```
... WHERE type_cd >= $ACT_RELATIONSHIP_TYPE_COMP_low
      AND type_cd <= $ACT_RELATIONSHIP_TYPE_COMP_high
```

- Dumb hacks won't be impossible.

This is going to be a major thinking task.

R-MIM Classes

Diminishing time and energy prevent me from making many examples about R-MIMs. By and large, R-MIMs are just constraints on RIM object graphs. So, every R-MIM class would in Java be an extension of a RIM class. For instance, Order is an constraint on Act:

```
interface Order extends Act { }
```

So now the question is whether and how the constraints should be checked. The constraint language that I am pushing in HL7 is quite close to Java; however, there is some bias towards OCL, which then would have a lot of translation work. However, some of the constraints are expressed in an even more concise form. For example, An Order is an Act where Act.moodCd() implies "MoodCode.order".

So, assume that all RIM objects had a common method:

```
BL isValid();
```

Then something could run the test operation on the entire instance graph (as far as it is loaded in memory or as far as it should be in memory ...). The isValid method body would be generated by the constraints, for instance, in our case:

```
class OrderImpl implements Order {
  BL validate() throws ConstraintViolationException {
    return getMoodCd().implies(MoodCode.ORDER);
  }
}
```

Let's say that other constraints would be that an Order has to have at least one Patient and one Author:

```
class HealthOrderImpl implements Order {
  BL validate() throws ConstraintViolationException {
    return getMoodCd().implies(MoodCode.ORDER)
      .and(getPatricipation()
            .exists(new Criterion {
                        BL test(Participation x) {

x.getTypeCd().implies(ParticipationType.PAT)
                        }
                    }))
      .and(getPatricipation()
            .exists(new Criterion {
                        BL test(Participation x) {

x.getTypeCd().implies(ParticipationType.AUT)
```

```
                    }
                }));
  }
}
```

One might say that special classes should not be generated for R-MIM objects, in which case those constraints would live in some metadata structure. However, I suppose that people would expect to see R-MIM classes implemented perhaps even more than they would expect RIM classes implemented. These R-MIM classes, like our HealthOrder could then have a nicer, more direct interface that exposes the Patient and the Author directly:

```
interface HealthOrder extends Order {
  Participation getPatient();
  Participation getAuthor();
}
```

I suppose that this may not only be nice for people, but it may also be the only way to use these classes and adapt them to legacy systems or even more recent systems. For instance, if company XYZ already has a Java implementation in which an Order has a Patient and an Author, then it would be inefficient to build the union of those for a HL7/Java interface that only knows Participation and then to have an HL7/Java component use a select method into the Participation set to find the Participation to find the Patient.

HL7 Messages

I am not going to show any XML at this point, because the HL7 XML is not even fully defined yet, and there will be at least one major overhaul from anything you might see today. But at any rate, an HL7 MESSAGE is an embodiment of an INTERACTION and an interaction is a tuple <SA, RA, TE, MT, RR> where

SA - sending application role
RA - receiving application role
TE - trigger event
MT - message type
RR - receiver responsibilities

And roughly this translates in two interfaces, one per application role and the interaction being a method of the receiving application role and the receiver responsibilities being some lofty description of what the sender can expect the receiver to respond (at some point in time, not necessarily immediately.) So, for example, let's take the interaction

```
Laboratory Observation Order Activate, Fulfillment Request,
Tightly-coupled (POLB_IN002121).
Sending application role:
   Laboratory Observation Order Lifetime Placer Tightly-coupled
   POLB_AR002131
Receiving application role:
   Laboratory Observation Order Lifetime Fulfiller Tightly-coupled
   POLB_AR002141
Trigger event:
```

```
   Laboratory Observation Order Activate, Fulfillment Request
   POLB_TE002120
Message type:
   Laboratory Observation Order Activate Tightly-coupled
   POLB_MT002101
Receiver Responsibilities
Reason                                          Trigger Event Interaction
Indicates that the request has been refused. POLB_TE002140
POLB_IN002141
Indicates that the request has been accepted POLB_TE003130
POLB_IN003131
and provides details of the plan to fulfill it.
Indicates that the order has been accepted,  POLB_TE004130
POLB_IN004131
```

And this provides the resulting event.

(I tried to speculate about what the interface to this could be on a high level and I deleted all my speculation as premature and more confusing than enlightening.)

We said that we will eventually render an object graph in an XML message. This would be done by just calling the general XML marshaller on the right object representing the message, i.e., we often have the whole thing begin with a Control_event structure:

```
Control_event myEvent = ...;
XMLOutput.write(myEvent);
```

and off goes the thing into some XML utility behind the XMLOutput.

There is lots more to think about, but I believe this is a useful beginning.