HL7 v3.0 Data Types Specification - Version 0.95

Table of Contents

HL7 v3.0 Data Types Specification

Version 0.95

Gunther Schadow
Regenstrief Institute for Health Care

Abstract

This document is a proposal for a complete redesigned set of data types to be used by HL7. Whereas in version 2.x data types where considered "formats" of character strings that would appear in HL7 data fields, this proposal assumes a more fundamental position: data types are the constituents of all meaning that can ever be communicated in messages. In HL7 v2.x, data types where defined *a posteriori* on an as-needed basis. Conversely this redesign defines data types *a priori* searching for fundamental semantic units in the space of all possible data types. This redesign work is heavily based on experiences with HL7 v2.x.

Data types are defined for (1) character strings and display data, which accomodates both character based text and multimedial data; (2) codes and identifiers for concepts and instances both of the real world and of technical artifacts; (3) all kinds of quantities including integer and real numbers, physical measurements with units, various kinds of time. Data types are classified (generalized) in various ways with respect to certain properties of interest.

A number of issues have been identified to be equally applicable to many if not all data types. Intervals (of ordered types), uncertain information, incomplete information, update semantics, historic information, and general annotations are defined as generic data types, that can be used to enhance the meaning of any other type. Although this type system is precisely defined, it has a lot of flexibility not found in many other type systems. Precise conversions are defined between types so that data of one type can be used instead of another if there is a conversion. As a special case, character string literals are defined for most types which allows an instance of composite types to be sent in one compact character string.

1 Introduction

This document proposes a redesigned system of HL7 data types to be used for HL7 version 3. It is the result of a task force group spawned off Control Query at the San Diego Meeting in September 1998. Since then, that group has been meeting in weekly phone conferences, chaired by Gunther Schadow. The following people (mentioned in alphabetic order) contributed to this endeavor: James Case (University of California, Davis), Norman Daoust (Health Partners), Laticia Fitzpatrick (Kaiser Permanente), Mike Henderson (Kaiser Permanente), Stan Huff (Intermountain Health Care), Matt Huges, Irma Jongeneel (HL7 The Netherlands), Anthony Julian (Mayo), Joann Larson (Kaiser Permanente), Randy Marbach (Kaiser Permanente), John Molina (SMS), Richard Ohlmann (HBO & Company), Larry Reis (Wizdom Systems), Dawid Rowed (HL7 Australia), Carlos Sanroman, Mark Shafarman (Oacis Healthcare Systems), Greg Thomas (Kaiser Permanente), Mark Tucker (Regenstrief Institute), Klaus Veil (Macquarie Health Corp., HL7 Australia), David Webber, and Robin Zimmerman (Kaiser Permanente).

This task force planned to conclude its work by January 1999. Although we made tremendous progress due to the commitment of the task force members, we were not completely finished. By January (Orlando meeting) we were about 80% finished. By April 1999 (Toronto), we have about 90% of the work done. As usual, the last parts of a project consume the most amount of time and energy. However, all data types are defined by now and the remaining work is to polish and refine.

This report is divided into two major parts. (1) The remainder of this introductory section explains the concepts and ideas that govern this proposed system of data types, while (2) the sections 2 through 5 will define the data types in detail.

This document was compiled from the notes of the twentyfour (???) conferences. The conference notes where issued in Hypertext (HTML) and publicly available for browsing (http://aurora.rg.iupui.edu/v3dt). In the notes I heavily utilized the unique advantages of the hypertext medium, namely the ease by which one can follow cross references. It so happened that general concepts and detailed definitions were mixed together as they came up in the conferences. Hyperlinks have been an invaluable tool to recall definitions and explanations from earlier notes and to show how ideas evolved over time.

This report is written as Hypertext too, but it is delivered to the general HL7 working group as a paper document, which required to bring the material into a systematic order. However, the division into a first part, explaining the overall concepts, and a second part, defining the data types in detail, is problematic, since the usefulness of the general concepts are illustrated only by how those concepts are actually used in the definitions of the data types. The definitions of the data types, however, depend on general rules. Thus the reader faces a kind of "hermeneutic circle", where one has to know about the first part before one can fully comprehend the second part and vice versa. The Hypertext version of this report contains numerous forward and backward links, which, in the printed form appear as cross references to page numbers in square brackets.

This ordering of the material comes in handy for the "impatient reader" who can explore everything just by following cross references. The reader who wants to see just some actual type definitions can use the index [p. 190] and directly proceed to the types he or she is interested in. The reader who wants to read through all the data type definitions can directly proceed to the sections of the second part [p. 33] and, if necessary, follow links back to the explanation of general concepts. Those who want to read through all of the text from the beginning can start with the general concepts and will be guided forward to the points where each concept is actually used.

A final word of acknowledgment. Many of the great ideas reported here are born in numerous and intense discussions that Mark Tucker and I had before and after the conference calls. Without Mark Tucker, this whole type system work would have never evolved to a useful state. I also want to acknowledge Mark Shafarman, whose great support was (and continues to be) vital for linking our ideas back to the HL7 organization, which we wanted to serve. Without him, our ideas might never have been able to touch ground. And last but most, I want to acknowledge Clem McDonald who keeps Mark Tucker and myself going by providing us with "fuel" and time to engage in HL7 work.

1.1 Goals

The overall goal of this redesign project has been rationalization and simplification of the HL7 data type system. This project is inspired by the tremendous redesign project "version 3" that guides HL7 into a competitive future. It starts with the observation that the number and complexity of HL7 v2.x data types has increased almost exponentially over the first 10 years of HL7 (from approx. 10 to 50 types) The reason for that explosion of types was new requirements that came up only in the recent years but were not anticipated by HL7's "founding fathers" who designed the data types system in 1988.

New requirements that we learned about in the version 2 period of HL7 had to do with the discovery that data in health care (and business in general) is not as clean as we thought at first. For example, the history of the TS data type shows the struggle with quantities that are imprecise in the real world and that all real world information is uncertain to some extent. Information may be wrong and needs to be updated, and most information items may change over time and we may have to keep track of the history (recent XAD changes initiated by Susan Abernathy with the National Immunization Program). Many data elements turned out to have more facettes to them than was expected, which lead to various X-variants of preexisting data types. New technology changed the way we think about telecommunication (TN-XTN) and formatted text (ST, TX, FT, HTML, SGML, RP, ED).

New requirements to an existing data type system must be met by modifications to the existing data types or by inventing new ones. In HL7 this sometimes lead to minor changes that could well be reverted later (TS). Sometimes the changes were felt so radical that the changed types were given other names (e.g., XPN, XAD, XTN, etc.). Over time the number of types grew and it became hard not to lose oversight.

In some ways, however, the old HL7 data type system was inherently flawed. The CM type, for instance, became a pain over time and we are still struggling to get rid of this undefined composite type. We just had too many data types for free text (TX and FT, recently growing to ED, HTML, etc.) and those arbitrary multiplicities multiplied the types that depend on free text, such as CE and CF. Such types as PN and AD were not designed under an international perspective.

The most deep flaw in HL7’s concept of data types was a wrong conceptualization of what a data type is. Data types were considered mere "formats" of data elements. This notion of a "format" is based on a focus on external representation (as character encodings) rather than on internal meaning. Thus data types where supposed to be constraints on character strings that would appear in data fields. This notion was in part supported by experience with programming languages that had a poor and weak type system, such as COBOL, BASIC or PL/1, that were widely used in business application programming.

Computer science, however, developed a much stronger concept of data types. Data types are now understood as the basic constituents of all meaning that can be processed with computers. The ALGOL family of programming languages (Pascal and MODULA 2) has a very strict data type system. At the same time their data types are extensible. New semantic entities were created by programmers through defining new types. Object oriented languages such as SMALLTALK, Eiffel, C++, and recently Java, have further elaborated this approach of creating new domains of meaning by defining types with operations. Common LISP and Scheme show a very well defined type system with emphasis on the semantics of types rather than representation.

From a deep collective understanding of HL7 version 2.x and its problems and from considering modern lessons of computer science, we can formulate specific goals and pathways of how to redesign a system of data types that not only improve the old one, but would also serve better in the future that may come with requirements that none of us may be able to conceive of today.

Semantics first

Data types are the basic building blocks of information exchanged in messages. Information is exchanged in the form of signals which are ordered according to lexical and syntactical rules. These signals are exchanged to convey a meaning (semantics) and to eventually serve a purpose (pragmatics). Therefore, data types must have a precisely defined semantics that is unambiguously related to their syntax (including the rules for building lexemes).

Usefulness and reuseability

The basic set of data types must be equally useful for all HL7 technical committees. This means, the data types must be meaningful enough so that the technical committees can use them directly as the data types for the attributes of their information model classes. It also means that the basic set of data types must be reusable for many purposes and should not be too highly specialized. This does not preclude a highly specialized data type to be defined by a technical committee that uses it.

Coherence

The set of all data types should be coherent. There should not be two or more competing data types for a certain use case. The relationships between the data types should be well defined. This means that data types should be organized similarly to the organization of domain information models (DIM) in the reference information model (RIM). The RIM and RIM harmonizations make sure that the DIM classes are in a close relationship and that there are no competing alternatives to express the same information in different ways.

Minimality

From the coherence requirement it follows that the number of data types in the set should be minimal. There should be just as many data types as there are independent basic semantic concepts to support. The lower boundary of minimality is that each data type should have a well defined semantics on a level that is relevant to the application domain of HL7. For example, we could have only one data type "string of bits", but bits do not have a generally relevant meaning on the application level of HL7.

Stability

It follows from the reusability requirement that every basic data type will be used by many classes and attributes of almost every technical committee. It becomes extremely difficult to coordinate changes to the data types and to estimate the effect that those changes would have on the many different areas in which the data types are used. Therefore the set of data types must be designed for high stability.

Completeness

Usefulness, reusability, coherence and stability can be achieved by aiming for maximal completeness *a priori*. This means that the data types of each basic semantic area cover that area to every logical extent conceivable by the time of design. Conversely completeness *a posteriori* would only make sure that every current concrete use case is covered by the design. Stability can only be achieved through aiming for complete coverage of every conceivable current and future use case.

Simplicity

The data types should be as simple as possible to ease implementation and use. This does not mean oversimplifications or neglect of requirements. Simplicity does not mean that the definition of the types would go with just a few simple words, because complete definitions are necessary for interoperability. Simplicity, however, does mean that exceptions, duplications, and dependencies are kept to a minimum. Simplicity mainly means that the type system should be easy to use and that it should prevent the user from making mistakes as much as possible. Simplicity also means that mistakes can be clearly seen as mistakes and prevented or fixed. Mistakes should not be hidden by imprecise definitions.

1.2 Methods

For our design of HL7 data types we can build on two kinds of prior knowledge and experience. There is more than ten years of experience with data types in version 2 of HL7 and there is more than 40 years of experience with data types in general computer science. In this proposal we will try to maximize leverage of these two rich sources of knowledge.

The redesign of data types is done in a top down fashion. We approach every semantic field by trying to understand what goes on. This understanding flows from experience and the identification of actual and possible requirements. But experience can only refer to the past. To reach stability and conciseness, we have to develop a precise semantic model that defines what exactly a type should mean and how it should be used. This definition is necessarily "theoretic" rather than practical, but it is meant to serve current and future praxis, not just academical curiosities. When the semantics is clearly enough defined, we proceed with specifying the structure of the types, i.e. their "abstract syntax".

We generally stop defining types at the abstract syntax level and we do not define specific mappings to XML, CORBA or other implementable technologies as part of this redesign work. This mapping to implementable technologies is a task of the Implementable Technology Specifications (ITS) prepared by special groups who focus on those technologies. However, many of the participants in this task force group know pretty well the the pain of implementing bad specifications, and some of us are part of the initial ITS definitions for XML and CORBA. Thus we do not neglect the actual implementation constraints. We will also continue to work on the ITS specifications as well as we will help the domain technical committees work with the new types.

ITS definitions of the data types should take into account not only the abstract syntax definitions but most of all the semantics and requirements of each data type. This is of utmost importance since the abstract syntax that we identify here is *not* absolutely normative. Variations in the abstract syntax definitions given here are allowed to make use of features that are available in a particular implementation technology. Variations of abstract syntax are permitted as long as the semantic features of the data types are all mapped to and preserved in the ITS.

Although we define data types top down, we will make sure that for every old HL7 v2.x data type there is at least one appropriate v3 data type. The mapping of types between v2.3 and 3.0 will be shown in an appendi [not done yet]. Some of our outstanding actions items are to provide help to technical committees to migrate to the new data types. Since no data types are assigned in the RIM so far and no durable messages specifications have been produced, this migration does not require any changes to actual version 3 specifications.

The intention in doing this theoretical approach is **not** to enforce some home-grown dogma of information science on system developers. It can not be made clear enough that through the type system proposed in this report, HL7 interfaces will not enforce new functionality on information systems. This type system aims in supporting new requirements, such as conveying uncertainty of information, but it does not force anyone to implement all of the features that it supports. We have defined a methodology called "implicit type conversion", to add enough flexibility to build bridges between systems that do have advanced features and those systems that do not have or need those features. We make sure that a sender can say all the detail that he wants to say about data items (not more and not less) and that the receiver can find as much information in a message as he can digest (not more).

1.2.1 Analysis of Semantic Fields

Guttman (1944) and Stevens (1953) identified four categories of data. Their classification coined the methodology for all sciences including biology, medicine, and psychology. Guttman and Stevens identified four *scales* on which we perform *measurements* or observations: (1) the nominal scale, (2) the ordinal scale, (3) the interval scale, and (4) the ratio scale.

We observe qualities on nominal scales. A nominal scale is a collection of all possible outcomes of an observation with no particular order. For example, gender, colors, or diagnoses are determined on nominal scales.

We have an ordinal scale when we can sensibly arrange the set of possible outcomes of an observation in an order. For example the NYHA classification of heart failure or tumor stagings are ordinal scales. We can determine the stage of the disease, we can tell the worse condition from the better, but we cannot measure distances, i.e. we cannot say that the step from NYHA I to NYHA II is as big as from NYHA II to NYHA III.

Interval scales are ordered quantitative scales, where you can measure distances (intervals) between two points. The paradigmatic example are the temperature scales Fahrenheit and Celsius. It does, however, not make sense to say 100 degree are twice as much as 50 degrees. However, the concept of the absolute zero temperature allows to make those decisions on the Kelvin scale (a ratio scale).

For an information standard in medicine it would be appropriate to reflect these fundamental categories of scientific observations. However, there are some problems with this classification.

- You can artificially try to upgrade the scale property. For instance, you can define an arbitrary order in qualitative observations (e.g., for gender: male = 0, female = 1).
- It often depends on the scope of the observation how you classify it, e.g., you can classify colors in any of those scales depending on what you think colors are (qualitative observations, up to wavelengths of visual light).
- The distinction between ratio and interval scales seems artificial because a simple translation of temperatures to the Kelvin scale is all that makes the difference.

Common sense might justify to distinguish qualitative from quantitative observations, although the color example shows that even the boundary between qualities and quantities can be blur.

We can further distinguish between observations that are discrete and those that are continuous, but again those are not precise categories. Many qualitative observations are continuous (i.e. color) but continuous qualitative observations are best understood by quantization. For instance, color can be quantized by wavelength of visible light, which is a scalar (a one-dimensional scale). But quantization can involve more than one dimension, as the color example shows: the RGB color quantization is a three-dimensional vector of numbers representing the intensity of the colors red, green and blue.

Since qualitative and quantitative, discrete and continuous observations are important in science as well as in everyday life, we can distinguish the field of discrete qualities and the field of quantities, both discrete and continuous. We will later have to show how to express continuous qualitative observations.

There are other important kinds of information not covered by the Guttman/Stevens classification: text. Text, is not just an abstracted observation and does not fall into the distinction between qualities and quantities, discrete and continuous. Text is chunks of information that are ultimately exchanged between humans. Computers and automatic messaging may be used to exchange text, but after having been entered by a human user, text is passed through essentially unchanged to be displayed to another human user. Text can express many observations, but this information content is not unlocked for the purpose of messaging and computer processing.

Text does not only include letters, words and sentences of natural human language, but can also be graphics or pictures (still or animated) or audio. Also, the same information content of natural language text can be communicated in written (characters) or spoken form (audio). Thus, we distinguish the field of textual information. Since one property of text data in messaging is that it is passed through unchanged and uninterpreted and without respect to the destination or purpose, we can subsume all other uninterpreted (encapsulated) data in the category of text.

There are thus three major areas of information that we identified by contemplating the broad field of all information. Those areas are pictured in Figure 1 [p. 9] .

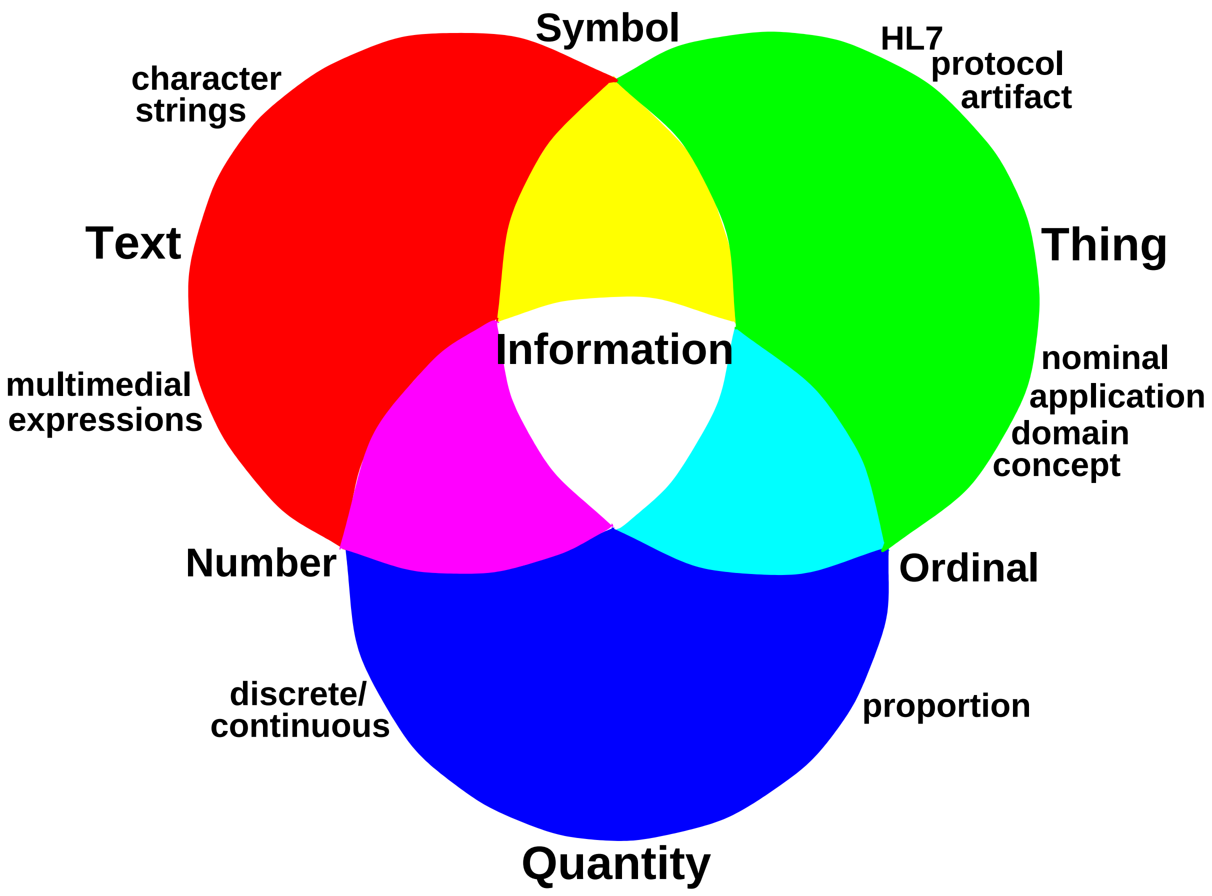


Figure 1: Phenomenology of Information.

Information usually consists of all three moments, text, thing and quantity: Information is always represented in some textual form; information is about things and concepts, which may have quantitative properties.

When talking about things, we have to use symbols to label the things and concepts we are talking about. Symbols are a form of text. However, the reverse is also true: text consists of things, i.e. letters, graphemes, or glyphs, that we recognize as distinguished concepts. Thus symbols are at the an area between text and things.

Likewise, numbers are represented through digits, which are characters, that is text. On the other hand, on computers, all text is stored in the form of binary numbers and only character code tables or image maps allow us to interpret those binary numbers as text. Thus, numbers are at the area between quantity and text.

A similar overlap exists between quantities and things. By enumerating concepts in coding systems, we can assign an ordinal number to each concept. On the other hand, concepts can have essential quantitative moments, if there is an order relationship, such as, for instance, in military degrees.

Because everything seems so blur and the boundaries between those areas of interest are not clearly demarcated, because there is no information that would not contains each of the three moments to some extent, it is hard to come up with any honest classification. The method of *phenomenology*, developed by G. W. F. Hegel (1807) and 100 years later by E. Hussel (1906), is a much better approach to such a messy field, that has so many facettes to it. The phenomenologic method basically observes how the meaning of the concepts drift and how concepts are in opposition to each other but, at the same time, depend on each other.

In this data type redesign, we guided our attention by those three major moments of information without neglecting the overlaps. Thus, our exposition of defined types will consist of the three major sections about text, things and concepts, and quantities.

1.2.2 Form of Data Type Definitions

Having said that the essence of data types is their semantics not their abstract syntax, we now introduce how we present the definition of some of the semantics and of the abstract syntax of our types. We use type definition boxes. The following is such a type definition box. Text set in italics stands for the fields that will be filled out for every defined type.

name of the type			
a brief textual description of the semantics			
component name	**type/domain**	**optionality**	**description**
name $_1$	*type* $_1$	*optionality* $_1$	*brief description of component* $_1$
name $_2$	*type* $_2$	*optionality* $_2$	*brief description of component* $_2$
...			
name $_n$	*type* $_n$	*optionality* $_n$	*brief description of component* $_n$

Some data types are so fundamental that there are no distinguishable semantic components. For example, an integer number is a closed well defined concept that can not further be split into components. We call such data types **primitive** data types as opposed to **composite** data types. Note however, that complex vs. primitive are relative qualifiers. In some implementation technology a primitive data type may well be implemented as having some internal structure and what we define as a composite data type may well be implemented using a primitive of some programming language. What is essential is that the semantics will be covered undistorted.

Data types that are primitive in our system are defined using a simpler type definition box as follows:

name of the type
a brief textual description of the semantics
PRIMITIVE TYPE

We initially considered to reuse the UML modeling tools for data types. However, after some experiments we discovered an interesting dilemma with using UML. There are two possible styles to define data types in a UML class diagram. Both styles have in common that every type would be represented by one class box, labeled with the name of the type. The one style would list all the semantic components as attributes in the box. Those attributes would again be defined as having a data type. Thus the name of other data types would appear in the list of attributes, almost like foreign keys. Obviously there are relationships between types but those relationships are not made visible. Every data type’s class box would stand on its own.

The other style to model data types in UML would be to depict the semantic components as relationship lines drawn from the containing type to the contained type. The role label at the side of the containing type would be the name of the semantic component. This results in an interesting diagram with just tiny little class boxes that maintain abundant relationships with each other, a picture that resembles a spider’s web. It is quite difficult to navigate through those many relationships.

Although using UML for data type definition is an interesting exercise it does not contribute very much to understanding of the types. The main problem with using UML is, however, that it evokes the impression as if the structure of the data types were all that needs to be said about the types. But the opposite is true. The most important part of the type definition is the defining and explanatory text.

1.2.3 Generalized Types

We use a notion of generalized types. Types can maintain an inheritance relationship with each other. We explicitly allow (and use) "multiple inheritance". However, we did not (yet) use inheritance as a way to specialize subtypes from general super-types. Rather we go the other way. Abstract generalized types are used to categorize the concrete types in different ways. Thus, we can get hold of all types that have a certain property of interest.

For instance, we define the generalized type *Quantity* to subsume all quantitative types. This is used to define one type Ratio [p. 139] as a ratio of any two quantities.

We defined a data type Interval [p. 166] that is a continuous subset of any type with an order relation. All types with an order relation are subsumed under *OrderedType*. Note that not all quantities are ordered (e.g. vectors are not) and there may be non-quantities that have an order relationship (ordinals, e.g. military degrees).

This categorization currently is done *ad hoc* rather than systematically. We will at some point revise and validate this *ad hoc* categorization. For instance, it seems as if *Quantity* may be too broad a category as it should contain ordinals. Ordinals, however, should not occur in a Ratio. It is unclear whether interval scaled quantities may properly occur in a Ratio, although most people would not worry about that.

1.2.4 Generic Types

Generic data types are incomplete type definitions. This incompleteness is signified by one or more *parameters* to the type definition. Usually parameters stand for other types. Using parameters, a generic type might declare components (fields) of other not fully specified data types. For example, the generic data type Interval [p. 166] is declared with a parameter T. In this example, T can stand for any *OrderedType*. The components *low* and *high* are declared as being of type T.

Before you can instantiate a generic type, you must complete its definition. For example, if you want to use an Interval [p. 166] , you have to say of what *base data type* the interval should be, i.e. you have to *bind* the parameter T. Say, you want an interval of Integer Number [p. 135] . You would bind the parameter T to the type Integer Number through which the incomplete data type Interval becomes completed as a data type *Interval of IntegerNumber*.

You can complete the definition of a generic data type right at the point of instantiation. This means, that you do not have to define all possible types generated by the generic type in advance. For instance, given the generic type Interval [p. 166] and the ordered types

- Integer Number [p. 135] ,
- Real Number [p. 136] ,
- Physical Quantity [p. 141] ,
- Monetary Amount [p. 142] ,
- Ratio of Quantities [p. 139] , and
- Point in Time [p. 148] .

You can use intervals of all those base types without having an actual specification of all the specific types. The specification, what an Interval is, is given only once, generically. Whenever you have a new ordered type, you can build an interval from it and use that new special interval, without having to define the new interval type explicitly. Generic types are thus a more efficient way of type specification.

Generic types became most popular in C++, where they are called *class templates*. In the C++ notation the Interval type would be defined as:

```
template <class T>
class Interval {
  T low;
  T high;
  ...
};
```

this interval generic type can then be used as follows:

```
Interval<int>   eligibleRankingNumbers;
Interval<float> normalRange;
Interval<Date>  effectivePeriod;
```

Generic data types may have more than one parameters. E.g. a type could be defined as

```
template <class N, class D>
class Ratio {
  N numerator;
  D denominator;
  ...
}
```

which is actually one way of making constraints: with this generic type `Ratio<T, U>`, is would be clear that `Ratio<int, int>` would be a ratio of two integers (a rational number), `Ratio<float, float>` would be a ratio of two floating point numbers, and `Ratio<float, int>` would be a ratio of a float and an int.

Note: Our data type Ratio of Quantities [p. 139] , is **not** defined as a generic type. Ratio is just used here to make an example about what generic types are.

Generic data types can be used in a nested way. Suppose you want an Interval of Ratios of `float`s by `int`s:

```
  Interval<Ratio<float, int>> foo;
```

would be all you needed to do to instantiate that new type.

Note: We did not decide on using the C++ notation of generic types, it is just used here because many people know C++ templates and thus C++ templates are a good illustration for what generic types are and how they work.

We will define generic types using type definition boxes that look like this:

name of the type			
a brief textual description of the semantics			
GENERIC TYPE			
parameter name	**allowed types**	**description**	
parameter name$_1$	*parameter types*$_1$	*brief description of parameter*$_1$	
parameter name$_2$	*parameter types*$_2$	*brief description of parameter*$_2$	
...			
parameter name$_m$	*parameter types*$_m$	*brief description of parameter*$_m$	
component name	**type/domain**	**optionality**	**description**
component name$_1$	*component type*$_1$	*optionality*$_1$	*brief description of component*$_1$
component name$_2$	*component type*$_2$	*optionality*$_2$	*brief description of component*$_2$
...			
component name$_n$	*component type*$_n$	*optionality*$_n$	*brief description of component*$_n$

As you see, the section defining the semantic components of the type is preceded by the keyword "GENERIC TYPE" and a parameter section. In this parameter section, type parameters are defined that are used in the subsequent section to define the semantic components. The parameter section may define a smaller number of parameters than there are defined components. Usually generic types go with just one parameter, sometimes there are two of them (as in the above Ratio example).

Please confer to the definition of the Interval [p. 166] to see a real life example of a definition box for a generic data type. For the interval, there is just one parameter *T* defined. Both boundaries of the interval are of the same type *T*. Any ordered type may be bound to the parameter *T*.

1.2.5 Collections

HL7 v2.x used the word "repeating" to describe certain qualities of the definition of fields and segments. This reflected the observation that "repeated" stuff could occur multiple times in the message. However, obviously there must be a reason why someone would make the decision that a segment or a field is to be repeatable in a message. It turns out that there are different reasons to make that decision. It was never clear from the HL7 spec. what the meaning of repeatability was in every instance.

The stuff that could repeat was either a segment or a field. For the purpose of this discussion we will consider the v3 equivalent of a segment to be a *class*, whereas the v3 equivalent of a field is an *attribute*.

If segments repeated in v3 this expressed a relationship (with multiplicity "1..*") between classes. When fields were declared "repeatable" this expressed a relationship between an attribute and its data values. We will concentrate here on the relationship between attributes and data values rather than on inter-class relationships, although what we say here is equally valid for class relationships.

In general, when things end up being "repeatable" we have a **collection** of things.

Consider the example of Patient "telephone number" (tel) that was declared as a "repeatable" field in version 2. The meaning of this is obviously that a patient has several telephones, we usually say, a patient has a "set" of telephone numbers. The word "set" implies that (1) it would not be meaningful if a given telephone occurred twice, and (2) that the order of telephone numbers does not matter.

We can use those two criteria to sort out the field of all possible collections, as the following 2 × 2 table shows:

	unordered	**ordered**
no multiples	*set*	*
multiples	*bag*	*list*

The ordered sequence without multiples is marked by an asterisk since this case is rarely considered in the computer science literature. Actually we can construct the field of collections as a lattice (a tree like structure) rather than a matrix. In such a construct, the *set* would be the parent of both *bag* and *list*, and ordered without multiples would not occur.

set

a collection of elements with no notion of order or duplicate element values. The number of distinguished elements in the *set* is called the "cardinality" of the *set*. An example of a *set* is the available fruits on a menu of a restaurant, e.g., { apples, oranges, bananas }.

list (or *sequence*)

an ordered collection of elements where the same value can occur more than once at different positions in the ordered collection. The notion of a *list* can be constructed from the notion of a *set* if we extend each element of the *set* by a position counter (a positive integer number). The number of elements in the *list* is referred to as the "length" of the *list*. An example of a *list* may be the list of my favorite fruits, where the fruit I like more precede the ones I like less, e.g., the *list* (orange, apple, banana) can be represented as the *set* { (apple, 2), (orange, 1), (banana, 3) }.

bag

unordered collection of elements where each element can occur more than once (think of a shopping bag containing 3 apples, 2 oranges, and 5 bananas). A *bag* can be constructed from a *set* if we extend each element with an occurrence counter (e.g., a *set* that contains the elements { (apple, 3), (orange, 2), (banana, 5) } is a *bag*. The total number of things in the *bag* can be called the "size" of the *bag*, the total number of different items can be called the "cardinality".

There are, however, other types of collections we frequently find, including vector and matrix. Those collection types, however, can be constructed using the above three fundamental collections, *set*, *list* and *bag*:

vector or *array*

a *list* with a specific length. Every position in that list represents one "dimension" (of the vector) or one "field" of the array. A vector need not represent geometric points in the 3D space and elements of a vector need not be numbers. Vectors are just a quantitative restriction on the *list* kind of collection, i.e. where the *list* must have a particular length.

(The length of a *list* can be restricted in other ways, e.g. lengths that must be between 1 and 5, those things are not vectors.)

matrix

a *vector* of *vectors* or a two dimensional *array*. Matrices are used for vector transformations or to describe network structures. Images could be thought of a matrices, but this is not the only way to think of images. HL7 probably has not yet a use case for *matrices*, but that may change as the Image Management SIG will contribute new contents to HL7

It should have become clear that there are many types of collections and subsuming them all under the (weakly defined) notion of "repeated" and "repeatability" is not very helpful to clear up the meaning of a collection. We thus want to do away with language that speaks of "repeated attributes" in the MDF to promote clarity regarding what specific semantic flavor of collections is meant in each case.

In case of waveforms, where "repeatedness" became quite tricky in v2.x, we can now define a sample of an *n*-channel waveform signal as a list of *n*-dimensional vectors, where each vector stands for a particular sample point in time.

One question was always associated with collections in HL7: how do we update those collections? We can distinguish the following cases:

1. The elements of the collection have identity (given to them through technical instance identifiers [p. 84]). Thus we can change some values of those elements. For example, if we have a list of individual practitioners, and if one practitioner changes her last name, we can simply change the last name of that individual instance. The only requirement is that the list elements have identity.
2. The elements of the collection have no identity. Changing the value of any given element is replacing that value in the collection, which in turn means changing the collection itself. Although we could change the value of the third element of a list of numbers, the position of an element in a list does not determine its identity. In a *set* or bag of numbers there is no "third element". The the only update one can do with a collection of values without identities is to add or remove elements from the collection. Thus, the question boils down to: How do we change the collections themselves?

One solution is to allow a collection to be updated only through separate trigger events with explicit message structures that would specify exactly what would be changed in which way. While this strategy works fine for high level RIM objects, such as, Encounter_practitioner, Clinical_observations, etc. However, for things like "*set* of stakeholder phone numbers" it is a bit too much of a burden to define specific trigger events.

But even if we had a trigger event "change patient phone numbers" it is not clear how we would specify what exactly should be changed.

For v2.x the answer always was: you send a snapshot of the collection as you want it to be and the recipient could simply throw away whatever he knows and would remember only what you just sent. This somewhat works in situations with just one master information producer and several slave information consumers, but it is totally insufficient for collaborative information management. For example, my message could wipe out all the telephone numbers that you already know.

We will give a solution below, when we talk about update semantics [p. 29] .

1.2.6 The Meta Model

The Meta Model discussion has been deleted from this specification and can now be found in the HL7 version 3 *Message Development Framework* (MDF).

1.2.7 Implicit Type Conversion

Implicit type conversion was an integral part of the technology that powered the flexibility of HL7 v2.x. Without being aware of the concept, HL7 coincidentally had a form of implicit type conversion that proved invaluable, especially for inter-version compatibility or localization problems. For instance, you could promote a single data element to a "repeating" element (i.e. a *list* of the base element) and vice versa without causing interoperability trouble with prior versions. Likewise, you could cast a data element declared as a primitive data type in one version of HL7 to a composite data type in another version. And you could "append" components "at the end" of a type definition, all without causing HL7 agents of different versions to reject each other's messages.

However, in HL7 v2.x, implicit type conversion was not a stated rule, it was sort of a by-product of the way HL7 messages used to be encoded. Transfer to other technologies, like C++ classes in ProtoGen/HL7 and IDL interfaces in SIGOBT's work lost this convenience of the implicit type conversion. If we want to preserve that invaluable technical feature of HL7 v2.x, we must explicitly state the precise rules of implicit type conversion.

Type conversion is also called "type casting". If a more primitive type is cast to a more complex type we can call this "up-casting" or "promoting" the lower to the higher level type. If a higher level type is being cast to a lower level type we call that "down-casting".

Type conversion must be clearly defined by reasonable rules. The rules should transfer the semantics of the data as good as possible. Especially the rules should not merely be driven by the coincidence of representations. For instance, it makes no sense to cast an ICD-9 code 100.1 to a real number 100.1 just because their representation happens to be the same.

The easiest way to state the rule for type conversion is by using a conversion matrix such as exemplified in the following table. The rows show the type you have and the columns show the type you need to convert to.

Example type conversion matrix

	String [p. 38]	FreeText [p. 46]	CodeValue [p. 58]	CodePhrase [p. 74]	CodeTranslation [p. 73]	ConceptDescriptor [p. 72]	Integer [p. 135]	Real [p. 136]	PhysicalQantity [p. 141]	Ratio [p. 139]
String [p. 38]	N/A	promote to text/plain	if code system is known and string is a valid code in the system	promote to CodeValue first	promote to CodeValue first	promote to CodeValue first	if string is a valid integer literal	if string is a valid real number literal	if string is a valid measurement literal	is string is a valid ratio literal
FreeText [p. 46]	if media type is text/plain	N/A	try conversion to string first	try conversion to string first	try conversion to string first	try conversion to string first	try conversion to string first	try conversion to string first	try conversion to string first	try conversion to string first
CodeValue [p. 58]	use the code or other rule for creating literals	convert to string first	N/A	make a phrase with just one CodeValue	promote to a CodePhrase first	promote to a CodePhrase first	none	none	none	none
CodePhrase [p. 74]	make a literal?	convert to string first	take first CodeValue in phrase (cave!)	N/A	new translation with origin set to `NIL`	promote to CodeTransaltion first	none	none	none	none
CodeTranslation [p. 73]	make a literal?	convert to string first	convert to CodePhrase first	use the *term* component	N/A	make new ConceptDescriptor	none	none	none	none
ConceptDescriptor [p. 72]	use "orignial text"? make a literal?	use "original text" or convert to string first	if a specific code system is needed, see whether it is in the set of translations	down-cast to CodeTransaltion first	if a specific code system is needed, see whether it is in the set of translations	N/A	none	none	none	none
Integer [p. 135]	use integer literal	convert to string first	none	none	none	none	N/A	make a real from an int, precision is number of all digits in the integer	make a real first	use as the numerator, set denominator to 1
Real [p. 136]	use real number literal	convert to string first	none	none	none	none	round the real number to an int, cave: this may create pseudo-precision	N/A	use "1" (the unity) for unit	use as the numerator, set denominator to 1
PhysicalQantity [p. 141]	use real number literal	convert to string first	none	none	none	none	down-cast to real first	return the value, may throw exception if unit is not "1"	N/A	use as the numerator, set denominator to 1
Ratio [p. 139]	use ratio literal	convert to string first	none	none	none	none	down-cast to real first	convert numerator and denominator to real and then build the quotient	cast the ratio values to a real number, make a new unit as the ratio of units (if any)	N/A

As can be seen the conversion matrix is sizeable, even on a subset of our types. There are other ways to picture the allowed conversions. For instance in a directed acyclic graph, where every data type is a node and every allowable conversion is an arc pointing from the type you have to the type you need. The arc would be labelled by the conversion rule used.

Conversions can be concatenated to eventually convert between "distant" types. This process is guided by pre-formulated strategy rules of the form "convert to *T* first". In a directed acyclic graph representation, finding those strategies resembles finding the shortest way between two locations on a road map.

The matrix representation and the directed acyclic graph are equivalent, thus one can use either of those representations of conversion rules. Since the matrix grows so big, we will probably go with the graph, which is an action item for future work.

Type conversion matrices can be interpreted by computers quite easily. In C, for instance, the matrix would be stored as a two dimensional array of function pointers:

```
typedef (*conv_func)(void*, void**);

conv_func conv_matrix[MAXTYPE, MAXTYPE] {
  { NULL, t1tot2, ..., t1totN },
  { t2tot1, NULL, ..., t2totN },
  ...
  { tMtot1, tMtot2, ..., NULL },
};

convert(int ti1, void *vi1, int ti2, void **vi2)
{
  conv_func cnv = conv_matrix[ti1, ti2];
  if(cnv != NULL)
    (*cnv)(vi1, vi2);
}
```

In C++ one can do the same or one can use polymorphism to make the process more obvious. C++ even has its own rules of implicit type conversion using cast operators, which could be used to some extent. In Java the process is mostly the same as in C++, but function pointers are not available. The above example does not show how concatenation and strategic steps can be used to convert between distant types.

In order for conversion rules to be used, a receiver first has to know what data type he has in a given message, in other words, the receiver needs to know the message element type (MET) of any given message element instance (MEI). Only then can the receiver know whether or not the type needs to be converted. Implementable Technology Specifications (ITS) of this type system therefore must make sure that the receiver has all the data type information he needs. This is most simply achieved by sending explicit data type information with every MEI.

The XML encoding designed in summer ’98 and used in the ’99 HIMSS demo, for example, uses an XML-attribute "TY" and mentions the data type as the value to the TY attribute. For instance, the following two MEIs for a simple integer number and a ratio of a real and an int could appear in a message.

```
<foo ty="int">100</foo>

<foo ty="ratio">
  <numerator ty="real">10.23</numerator>
  <denominator ty="int">5</denominator>
</foo>
```

The receiver might expect `foo` to be a real value. Using the conversion rule *convert numerator and denominator to real numbers and then build the quotient* [p. 19] the receiver can convert the type he has to the type he needs.

Mark Tucker’s rule of minimal explicitness states that you only need to send TY attributes at a place where the actual type used diverts from the specification. However, deciding that is a lot of responsibility on the sender’s side. It is therefore safe to always send TY attributes. For the HIMSS demo we simply made it the rule that the sender must supply explicit data type information in TY attributes.

When generic types are used, the TY value only specifies the generic type. The type of the parameters is found where the value of that type is expected to be. Thus, regardless of what is otherwise decided, TY attributes are always required for the parameterized components of generic types.

Conversion rules must be carefully validated to prevent surprises. For example, suppose we had a generic data type "QualifiedInformation" that would allow to add some coded *qualifier* to any other *value*. The conversion rule would say: whenever you need a *T* and you get a qualified *T*, just take out the *value* part and do not consider the *qualifier* part. Now consider that one qualifiers, "`NOT`", would exist for negation. What would happen if a message element instance contained

```
<main-concern ty="QualifiedInfo">
  <value ty="ST">PNEUMONIA</value>
  <qualifier>NOT</qualifier>
</main-concern>
```

While the sender would mean that the "main concern" is **not** pneumonia the receiver would understand just the opposite! This shows that conversion rules have to be specified with great care. In this case, conversion to simply pneumonia should be prohibited, i.e., the conversion routine would either return no value or raise an exception.

1.2.8 Literals

In the example type conversion matrix [p. 18] many special conversions exist between Character String [p. 38] and any other type. This is because we want to define concise and nice looking string literals for many of the data types, whether primitive or complex types. String literals can be used in XML, for instance, to make the message more compact and human-readable.

Literals can be used to specify data type instances in character oriented encoding rules. It is good to have a single standardized form of literals to be used by different ITSs. Literals are useful for many ITSs, not just XML. For instance, SIGOBT did use character representations of most data types in their v2.x mapping of HL7.

Literals are not only useful in inter-system messaging but also when we discuss about the design of HL7 messages on a black board or in e-mail. Literals are much more handy than structured instance notations, such as XML. The guideline for the specification of literals is that literals are to be concise and easily understandable by humans.

1.2.9 Instance Notation

For the purpose of discussion and to be able to show examples of data types we will use an instance notation that is both, readable and concise. We do not use XML as an instance notation since XML is just too verbous, writing XML on a blackboard takes too much time, and the XML markup is too distractive for the human eye to find the real information to be conveyed in the example.

Our notation is borrowed from *Common LISP* and *Scheme*, a syntax also used in the XML world (DSSSL).

This instance notation has only five idioms

1. Atomic values (numbers, strings, symbols) are written in the usual character representation. Atomic values are separated by spaces, unless the spaces are contained within double quotes. For example

   ```
   1234.45
   ```

   the a number 1234.45

```
"hello world"
```

a string

```
foo
```

a symbol

2. Composite values start with an opening parenthesis and end with a closing parenthesis.

   ```
   ( ... )
   ```

3. Composite values may contain atoms or other nested composites.

   ```
   (foo :bar (nest :baz))
   ```

4. Composites always start with a symbol that denotes to the data type of that composite value. In the example above, `foo` would be the symbol of the data type.

5. After the type symbol, composites contain keyword-value pairs. Keywords are symbols that start with a colon (e.g., `:bar`). For example

   ```
   (CodeValue :value      "100.0"
              :codeSystem "ICD-9")
   ```

   would be a Code Value [p. 58] representing the ICD-9 code 100.0 for Leptospirosis icterohemorrhagica.

6. Symbols that start with a pound sign have special meaning. For instance, `#true` and `#false` would be two values for the Boolean [p. 24] type.

7. Collections [p. 14] are composite expressions whose first symbol denotes the kind of collection (i.e., `SET`, `LIST`, or `BAG`). After the collection type symbol the elements of the collections are enumerated. For example,

   ```
   (SET apple orange banana)
   ```

   a set of fruits, cardinality 3.

   ```
   (LIST orange apple banana)
   ```

   the list of fruits ordered by how much I like them, length: 3.

   ```
   (BAG 3 apple 2 orange 5 banana)
   ```

the shopping bag containing 3 apples, 2 oranges and 5 bananas, size: 10, cardinality: 3. Note that the bag notation uses alternated number-item-pairs.

The beauty of this instance notation is that it can be completely defined by just a few simple rules. Moreover, the examples can usually be understood without the reader having to be able to actively master the rules.

1.2.10 *Typus typorum*: Boolean

Let’s define the first real data type, a primitive type to start with. Which type could be a better starter than the Boolean type, the type of all types? A Boolean value can either be true or false. The Boolean is the smallest quantum of all information (1 bit) and yet all digital information is based on it. While Boolean values are the very basic values of all digital information processing machinery, the Boolean data type is useful even in the highest sphere of abstract data analysis. The Boolean type embodies the axioms of logic. This is a universality that only the Boolean type has.

The Boolean type is defined as follows

Boolean (BL)
The boolean type stands for the values of two-valued logic. A boolean value can be either *true* or *false*.
PRIMITIVE TYPE

Use cases for the Boolean type are all RIM attribute with the "attribute type" suffix "`_ind`" (indicators).

HL7’s position on Booleans used to be that of an ID data type with the special table that included only the values "Y" and "N". Since the follow-up data type for ID is Code Value [p. 58] , we could continue to serve the use case for Booleans with Code Value [p. 58] constrained to the "Y/N" table.

The reason not to continue with this habit is that Booleans are just so universally useful and by the way are the simplest data type of the universe. Boolean information items exist and are useful on virtually all levels of abstraction, so that it would be a move toward simplicity to define an explicit Boolean data type for HL7 to be used for all "indicators". It is so much more easy to use Booleans in program decisions, as the following example in a fictive programming language shows:

```
VAR
  X : BOOLEAN;

...

IF X
THEN
  (* X is true *)
ELSE
  (* X is false *)
END IF;
```

By contrast, dealing with an arbitrary Code Value [p. 58] requires to first check whether the code table used is the Y/N-table, then you would have to treat every possible case including that the given value is neither "Y" nor "N" (because there is no guarantee that the Y/N-table never changes, see below).

```
VAR
  X : CodeValue;

...

IF X.codeSystem == CodeSystem.Y_N_TABLE
THEN
  IF X.value == "Y"
    (* X is true *)
  ELSE
    IF X.value == "N"
    THEN
      (* X is false *)
    ELSE
      (* EXCEPTION: X is neither true or false *)
    END IF;
  END IF;
END IF;
```

Why would we not want to use boolean data types?

- For backwards compatibility to the Y/N table?
- Because Technical Committees might want to refine the table later?

Backwards compatibility to v2.x has never been (and should not be) the major issue for design decisions for v3.0. However, through type conversions we can actually allow for backwards compatibility. Thus, a Boolean would convert to a Code Value [p. 58] by using the Y/N table. Any Code Value [p. 58] with the coding system set to the Y/N table can be converted to a boolean.

Note: We should, however, not define a conversion from Integer Number [p. 135] to Boolean on the basis of 0 = false, 1 = true. While the Y/N table's semantics is clearly to represent Boolean values, the mapping of Booleans to numbers is not semantically suggested nor is the mapping style determined by semantics (e.g. one could map false to -1 and true to 0, or false to 0 and true to non-zero just as well).

Some people might think that using the Y/N table to capture Boolean semantics is more flexible, because one could later extend the table to cover other (exceptional) values. For instance, some might want to add the value P for "perhaps" and U for "unknown". Those two extensions to the Y/N table can be called "generally applicable", since they are conceivably valid for all cases where the Y/N table is used.

The programming example above shows why you just not want to extend a table used as a replacement for Booleans. Relying on Booleans means relying on one of the fundamental axioms of logic (*tertium non datur*), sneaking in a third code into the Y/N table would render this axiom of logic invalid, which means that every `if ... then ... else ...` statement would have to mutate into a `case ... of ... otherwise ...` statement.

Those "generally applicable" extensions of the Y/N table are not just a bad idea, they are also not necessary in the context of this data type proposal. The value "perhaps" is covered by all the mechanisms to define uncertainty [p. 174] , and the "unknown" exception is covered by the method to handle incomplete information [p. 27] .

Other people might still think that the Y/N-table should be used to allow for subsequent extensions. An example might be for the patient *death indicator*, where Y/true means the patient is dead and N/false means that the patient is alive. Now, one could make the case that a patient after the diagnosis of "brain death" might be kept in a vegetative state until some organ transplantation. This would be a status between live and death that neither falls in the category of uncertainty nor incomplete information. So, one might need to extend the Y/N table by "B" for "brain death".

Clearly, such extensions of the Y/N table could be made only at one point of use of the Y/N table, e.g., only the *death indicator* would use the Y/N table extended by "B" for "brain death". This means that *death indicator* no longer would be defined as a code from the Y/N table, but from a "death code" table. According to the MDF, the attribute type suffix "`_ind`" would have to be changed to "`_cd`".

If "death indicator" would have been defined as a Boolean in version 3.0 and later would have to become a code of table "death code" one could either simply change the data type definition between versions or, instead, add another field, such as "death detail status" if "death indicator" is true. Those changes in the use of the field do require RIM changes regardless of whether we used the Boolean data type or not.

If nothing else, a Boolean data type could help sharpen the analytic work of the committees, because it would be absolutely clear whether or not there can be other values aside from the two opposites represented by true and false.

1.2.11 Incomplete Information

In v2.x we had the special values **not present** (`||`) or **null** (`|""|`) that could be sent instead of any other value in almost every field in a message. The semantics of those special values were two fold (1) not present expressed that information was missing (2) null was able to remove existing information at the side of the receiver so that this information was missing afterwards. We will factor this "update" component out into update semantics [p. 29] below. Here we only deal with the representation of incomplete information. This means, NULL values do no longer automatically carry the notion of "deleting" or "overwriting" with them.

After having defined the Boolean, the type that underlies all information, we now define a data type called "No Information" as follows:

No Information			
A No Information value can occur in place of any other value to express that specific information is missing and how or why it is missing. This is like a NULL in SQL but with the ability to specify a certain flavor of missing information. The No Information type extends the value domain of any other data type unless explicitly forbidden by domain constraints.			
component name	**type/domain**	**optionality**	**description**
flavor	Concept Descriptor [p. 72]	**optional**	The flavor of the null value. Can be interpreted as the reason why the information is missing.

The "flavor" of the null value can be interpreted as the reason why the information is missing. For the time being we keep the list of possible flavors of null subject to open discussions. Reported numbers of different flavors of null values range between 1 (SQL) and 70 (reported by Angelo Rossi-Mori). If No-Information flavors are to be used in a standard way, we have to define a canonical systematization of flavors of null. The following table lists a number of canonical null value flavors plus additional flavors of null which still need to be systematized.

NI	*no information* canonical	This is the default null value. It simply says that there is no information whatsoever given in the context where the NI value occurs. The information may or may not be available elsewhere, it may or may not be applicable or known. The NI value can not be interpreted any further.
NA	*not applicable* canonical	The data element does not apply in a given context, e.g. an answer to "gestational age" for a patient who is not pregnant.
UNK	*unknown* canonical	The information may be applicable, but is not known in the given context.
OTH	*other* canonical	The information is known but can not be expressed in the required constraints. Most often used when a concept needs to be coded but the code system does not provide for the appropriate concept. Many code systems have an "other" entry (also called "not otherwise specified".) Terminologies should not themselves contain "other" entries [Cimino ??]. The null value of the OTH flavor can and should replace those "other" codes. **Note:** this flavor is **not** itself a "not otherwise specified" code for null flavors.
NASK	*not asked*	the person who should collect that information forgot to ask. Needs further systematization.
ASKU	*asked but unknown*	the person asked could not supply the information (why?) Needs further systematization.
NAV	*not available*	the person asked does have the information somewhere but not available right now (e.g. oh, I wrote down what the doctor said last time, but I didn't bring this piece of paper with me). Such data elements might be updated soon. Needs further systematization.
NP	*not present* special	The *not present* value is only meaningful within a message, not within a system's data base. The not-present flavor must be replaced by the applicable default value at the receiving interface. If no other default value is specified, a No Information value with the dafalut flavor *no information* is used.

In most cases, the No Information value with the default flavor *no information* is sufficient. So, if the flavors of null are deemed not useful for technical committees or implementors, they can simply assume no flavors to exist other than the default *no information* flavor (which would translate to an SQL NULL) and the special flavor *not present* which is only applicable for messages and is replaced by a default value at a receiving interface.

For example, consider the patient's date of birth is requested and we don't know the date of birth because the patient does not remember it. In that case we could send:

```
(Patient
   :date-of-birth (NoInformation
                    :flavor (CV :value "ASKU")))
```

In this example instance notation we will use the symbol `#null` to be equivalent with `(NoInformation)` with the implied default flavor *no information*.

Note that No Information is formally a composite data type, although it has but one component. We will list No Information under the category "primitive" anyway, since it is so fundamental to our type system. This is a very special data type anyway, since it will never be used in declaring attributes or data elements, but will rather extend every data type to provide for a consistent way to account for missing information.

Note that extended Boolean logic (e.g., three-valued logic) is supported using the classic Boolean data type with the implied domain extension offered by the No Information values. The third value of three-valued logic would be the No Information value (of any flavor.) The logic operators that apply in three valued logic are defined in the following tables:

Definition of logic operators in three-valued logic

NOT	
true	*false*
false	*true*
ni	*ni*

AND	*true*	*false*	*ni*
true	*true*	*false*	*ni*
false	*false*	*false*	*false*
ni	*ni*	*false*	*ni*

OR	*true*	*false*	*ni*
true	*true*	*true*	*true*
false	*true*	*false*	*ni*
ni	*true*	*ni*	*ni*

1.2.12 Update Semantics

Update semantics deals with the problem of what a receiver is supposed to do with information (or "no information") in a message. That information may be equal to prior information at the receivers data base, in which case no questions occur. But what if the information is different?

We can categorize the modes of updates in the following taxonomy:

1. IGNORE: Ignore the value all together
2. VERIFY: Verify whether the value supplied matches the prior value. If the values do not match, raise an exception.
3. REPLACE: Replace the value in the data base with the new value supplied in the message. Replace operations may be of the two more kinds:
    1. REPLACE VALUE: Change an old value to a new value
    2. DELETE: Change an old value to a No Information [p. 27] value (i.e. a null value).
4. EDIT COLLECTION: If the data is of some collection type, we can change the collection in specific ways depending on the kind of collection:
    1. A *set* can be updated in one of the following ways:
        1. include elements: build the union of the set and another set.
        2. exclude elements: build the difference of the set and another set.
    2. A *list* can be updated in one of the following ways:
        1. add element
            1. append
            2. prepend
            3. insert at given *position*
            4. insert at element with given *value*
                1. before
                2. after
        2. replace (either replace with new value, or set to no information)
            1. by position
            2. by value
                1. first occurrence
                2. last occurrence
                3. *n*-th occurrence
                4. all occurrences
        3. delete element entirely, changing the positions of all other elements after the deleted one.
            1. by position
            2. by value
                1. first occurrence
                2. last occurrence
                3. *n*-th occurrence
                4. all occurrences
    3. A *bag* can be updated in one of the following ways
        1. include elements: build the union of the bag and another bag.
        2. exclude elements: build the difference of this bag and another bag.

3. exclude all of elements of one kind: e.g., if a bag contains 5 apples and 3 oranges, you could exclude all oranges without having to know that you actually remove 3 oranges.

In principle, the update mechanism will send an update action code along with each message element instance (MEI). The update action code should be part of the MEI meta model.

It turns out that updating a *list* is the most difficult task to do, since positions are relevant in the list. The problem is concurrent updates; you never know exactly what the list looks like at the receiver's system when your update message is being processed. For example, if you think the list is `(LIST A B C)` and you want to insert an element `D` to come before `C` you may send an update expression

```
  (INSERT-AT 3 'D)
```

to insert `D` at position 3 (and shift `C` to position 4). However, if someone rearranged the list to `(LIST C B A)` just before your update message arrives, the receiver would insert the `D` between `B` and `A` and you would cause the list to change to `(LIST C B D A)`.

If what you really wanted was to insert `D` before `C`, you should have sent the update expression

```
  (INSERT-BEFORE 'C 'D)
```

which, at the receiver's side would update `(LIST A B C)` to `(LIST A B D C)` but also `(LIST C B A)` to `(LIST D C B A)`.

The sender of an update message has to be very sure whether he wants the new element to appear in a particular position within the list or in a particular sequence relationship with another element of the list. Concurrent edits to the same data at the receivers side can render the sender's assumptions invalid.

Conversely, with *sets* concurrent updates are not a problem at all, because the only thing to do with a *set* is adding or removing values to and from the *set*, which is independent on the prior contents of the *set*. For example, if you add a telephone number to a set of telephone numbers, it doesn't matter whether or not that telephone number is already known, since there are no duplicates of the same value in a *set*. Likewise, if you remove a bad telephone number from the *set*, you can do so no matter if the number was element of the *set* before. Also, there is no ordering that could get messed up, nothing to assume before the update, so no assumptions can be invalidated through concurrent updates.

Updating a *bag* is equally straight forward. If you want to add 2 apples into the bag, you do that without having to know how many apples where there before. If you want to remove 3 oranges, you can do that, no matter how many oranges were there before. Note that removal of items from a bag does not mean here that you want to get hold of those items, you just want them to disappear from the bag. Thus, if there are no more oranges left in the bag to be removed, your removal request is satisfied without changes.

For the technical committees this means that a *list* collection semantics should only be chosen if the order **really** matters semantically from the perspective of pure abstract application logic. If the order probably is not important enough to justify the headache around concurrent updates, the committee should choose the *set* or *bag* flavor.

Selecting *set* and *bag* semantics should always be encouraged. A *set* is often exactly the right kind of collection from the perspective of pure abstract application logic. Most collections, in practice, are *sets,* while *bags* are quite rare.

If the collection element type is a class, such as Condition_node, and a ranking is important, the ranking could be represented explicitly by a ranking number rather than implying *list* semantics on some association, even though it is possible in UML to assume list semantics of an association.

Also note that there are *partially ordered* collections that often capture the application logic much better than totally ordered lists. Partially ordered collections are collections where elements may have the same ranking, so that you can not always decide whether one element has higher rank than another.

2 Text

2.1 Introduction

All information can be expressed by sequences of bits, this is the fundamental new discovery that started the era of digital information processing. Written text consists of characters and characters are by themselves expressed as sequences of bits. Eight consecutive bits are called *octets* or *bytes*. Although we usually identify one byte with one character, this identification is not an eternal law of nature and we have to distinguish bytes from characters.

The ease by which we express characters as bytes and bytes as characters is due to the success of the *American Standard Code for Information Interchange* (ASCII) [ANSI X3.4]. Most computers interpret bytes as characters according to the ASCII code. But this does not mean complete peace of mind. On the one hand, although ASCII is by far the most important character code, there is another one: EBCDIC.

On the other hand, ASCII does not define sufficient characters to meet the needs of non-English languages. ISO 8859-1 defines an international extension to the ASCII code that fits most languages of the world that use Roman charcters (Latin-1). However, there are numerous other such extensions. And there are numerous other languages, including Greek, Russian, and Japanese.

We cannot even count on the truth that one character is expressible in one byte, as we learn from Japanese and Chinese character sets that have way more characters than would be enumerable with just 8 bit.

The solution to the Babylonian coding chaos seems to be the Unicode standard [ISO/IEC 10646, Unicode (http://www.unicode.org/)]. Unicode is a **character set** that covers all languages of the world, with even the rarest being added in upcoming versions of Unicode.

Unicode seems to be accepted in all major language communities including America and western Europe, Russia and the three countries China, Korea, Japan that were so often left alone with their character coding problems. China, Korea and Japan have submitted to the Unicode a jointly compiled unified character set, called "Han", which includes more than 20000 characters. Of course, those many characters can not be enumerated with only 8 bits, thus, one Unicode character uses more than one byte.

2.1.1 From Characters to Strings

While most programming languages define data types for single characters, HL7 messages did not use single characters as opposed to character strings in the past and probably will not do so in the future. A single character is on a too low level of abstraction. There is no clinical or administrational information expressed in one character that stands for itself. There are single character codes, such as the "sec code" consisting of the symbols "M" for male and "F" for female. Those characters "M" and "F", however, do not stand for themselves but for some other meaning. Therefore we will not need a data type for single characters.

2.1.2 Display Properties

A character code like ASCII, ISO 8859, or Unicode codifies only characters, i.e., the basic graphemes from which written language is constructed, regardless of the style-variants of characters. Often we are only interested in transmitting the semantics of a few words or sentences. But sometimes we want to enhance the expressiveness of text through an altered appearance of characters. One can modify font family (e.g., Times Roman, Helvetica, Computer Modern), font style (e.g., roman, italics, bold), font size (e.g., 8 pt, 10 pt, 12 pt), alignment (e.g. subscript, superscript) or any other display properties.

The question is, for what use cases we need only plain character strings and when do we need control over the appearance of the characters?

When a data field contains only one or a few words, we will probably not need control over appearance. However, who is to say how many words may appear in a given data element of type string? And what is the exact limit of words that do not require formatting? Clearly the length of a character string is no good criterion for whether formatting is required or not.

Instead we need to look at fine semantic nuances to find the answer: A string that encodes a value from a code table (e.g., "M" or "F") will not need formatting. A string that encodes a person's first name or address will not need formatting too. These informations, code symbol, person names, or address are readily conveyed only in the characters. To make this more clear. I always refer to the same city Indianapolis, regardless whether I write its name in bold letters (**Indianapolis**), italics (*Indianapolis*), underlined (Indianapolis), or any combination of those or other display properties.

Conversely, controlling appearance of text will be useful in those data elements whose purpose it is to be shown to human users. Even of only two words, we sometimes want to emphasize one word by underlining or emboldening it. There is no reason to prevent formatting for those data elements that are placeholders for free text. Thus we have to distinguish between formalized information and free text to find out when we need control over appearance.

2.1.3 Encoding of appearance

The format of a text is encoded in three different ways:

1. through deploying certain intrinsic features of the underlying character code,
2. through specially reserved positions in the underlying characters code, or
3. through escape sequences.

Ad 1: The ASCII control character number 8 ("backspace") can be used to overstrike an already printed letter. Thus one can print the same letter twice or three times to yield an emboldened appearance on a simple typewriter or dot matrix printer. One can also print the underbar character over the previous letter to yield the effect of underlining. There are simple software programs that emulate the behavior of a typewriter to render this kind of simple formatting. For example, the UNIX "more" utility used to display online manual pages emulates a typewriter and some terminal devices have this emulation built in.

Ad 2: Many text processors use other control character in non-standard ways to encode the formatting of the text. For example if you look at the raw file of a Word Perfect text, you will find the words and characters interspersed with control characters that obviously encode the style of the text. The problem with this approach is that it is proprietory and not standardized.

Ad 3: *Escape sequences* are used by various printers and terminals. Originally, those were control sequences separated from the normal text by a leading ASCII character number 27 ("escape"), hence the name "escape sequence". But escape sequences have since been used in many different styles. In C string literals, TROFF, TEX and RTF we see the backslash character (\) introducing escape sequences. TROFF has a second kind of escape sequences started by a period at the beginning of a new line. HL7 version 2 also uses the backslash at the beginning and end of escape sequences. SGML uses angle brackets to enclose escape sequences (markup tags), but in addition there are other kinds of escape sequences in SGML opened with the ampersand or percent sign and closed with a semicolon (entity references).

From the many choices to encode formatted text HL7 traditionally used a few special escape sequences and TROFF-style formatting commands. Those HL7 escape sequences have the disadvantage that they are is not very powerful and somewhat arcane or at least outdated by the more recent developments. HTML has become the most widely deployed text formatting system, available on virtually any modern computer display. HTML has been designed to be simple enough to allow rendering in real time. Thus HTML seems to be the format of choice to transmit style-enhanced free text.

A considerable group of HL7 members also pursue using SGML or XML to define text, although the purpose to using general SGML or XML is slightly different from using HTML. Where HTML is used to control logical appearance of text, SGML is another way to structure information. Thus HL7 will use SGML as one of its message presentation formats. SGML in free text fields is so powerful and general, that it comes with the risk of not being interoperable. However we might want to allow for it in special circumstances.

It will be difficult to limit the HL7 standard to just one of the possible alternative encodings of appearance. There is an issue of backwards compatibility that requires to keep the nroff-style formatting of HL7’s FT data type. There is a tremendous and reasonable demand for supporting HTML, and we should not exclude general SGML and XML up front, despite the concerns for interoperability.

There are, in principle, two ways to support the multiple encodings of appearance. Either we define multiple data types, one for old FT, one for HTML and one for general SGML/XML, or we define one data type that can contain formatted text in variable encodings.

Defining multiple data types has the disadvantage that we need to decide at design time for one of those alternatives whenever a free text data element is defined. This decision is unchangeable at the time an individual message is constructed. In other words, technical committees would have to decide to use the old FT type here, the HTML data type there, and a simple TX type for yet another free text attribute. There is hardly any rationale for such a decision at design time of the standard.

Thus, the irrationality and inflexibility of defining multiple data types for free text seems to outweigh the conceivable advantage that a special data type might accommodate the intrinsics of some special encoding formats in greater detail and accuracy. Thus, we define only one flexible data type for free text, that can support all the techniques for encoding appearance of free text.

2.1.4 From appearance of text to multimedial information

Being able to format the appearance of free text adds a great deal of expressiveness. But having control over graphical appearance of text begs the question whether graphics, drawings and pictures should not also be considered part of free text, for "a picture says more than thousand words"? In human written communication, especially in business and science, we often use drawings to illustrate the points we make in our words. The technology to do these things on computers is available, HL7 only has to support it.

Another use for multimedial information is that this is the only way to capture the state of a text that precedes its typed form: dictation and handwriting. An HL7 message that is sent of from a Radiologist’s or Pathologist’s workplace will usually contain very little written information, but rather the important information will be in dictated form. Again, the technology to capture voice data, to communicate, and replay it is available on almost any PC now, HL7 only has to support it.

Two alternatives exist to support multimedial information in HL7. Since HL7 version 2.3, we can use the "encapsulated data" (ED) type. The ED data type is powerful enough to communicate all kinds of multimedial information. The problem is that it is a special data type that can only be used in data fields assigned to the ED data type. Currently none of the HL7 data fields is explicitly assigned to the ED data type, which considerably diminished ED’s usefulness despite its power.

The only way to use the ED type is currently in the variable data type field OBX-*observation-value*. While this serves the communication of diagnostic data that is in image or sound form, it is not generally usable. For any multimedial data we want to send per HL7 we have to pretend that it is diagnostic data even if it isn’t. If we want to send some descriptive drawing to an order, we have to pretend it’s diagnostic data and send it in an OBX. Furthermore, it is not even clear whether there will be a variable data type in HL7 version 3.

The honest alternative to support multimedial data would be to admit that any free text data can possibly be augmented or replaced by multimedial information. This means, we have to allow for multimedial data in any free text field, and thus, that free text and multimedia data share the same data type. This is not hard to do since one flexible data type was already required to accommodate the different encodings of text formats. We will call this data type "Display Data" and it is used for both free text and multimedia. Display Data will consist of a media descriptor code and the data itself. Applications will render the data differently depending on the media descriptor code.

Although it is technicallz convenient to merge character-based free text and multimedia data into one data type, the rationale of this decision is semantic not technical. Both, character based free text and multimedia data is information sent primarily to human beings for theiur interpretation. This conforms to the meaning of the word "text" as explained by Webster’s dictionary (http://www.m-w.com/home.htm):

Main Entry: **text**
Pronunciation: `’tekst`
Function: *noun*
Etymology: Middle English, from Middle French *texte,* from Medieval Latin *textus,* from Latin, texture, context, from *texere* to weave -- more at TECHNICAL
Date: 14th century
1 a (1) **:** the original words and form of a written or printed work (2) **:** an edited or emended copy of an original work **b :** a work containing such text
2 a : the main body of printed or written matter on a page **b :** the principal part of a book exclusive of front and back matter **c :** the printed score of a musical composition
3 a (1) **:** a verse or passage of Scripture chosen especially for the subject of a sermon or for authoritative support (as for a doctrine) (2) **:** a passage from an authoritative source providing an introduction or basis (as for a speech) **b :** a source of information or authority
4 : THEME, TOPIC
5 a : the words of something (as a poem) set to music **b :** matter chiefly in the form of words that is treated as data for processing by computerized equipment <a *text*-editing typewriter>
6 : a type suitable for printing running text
7 : TEXTBOOK
8 a : something written or spoken considered as an object to be examined, explicated, or deconstructed **b :** something likened to a text <the surfaces of daily life are *texts* to be explicated -- Michiko Kakutani> <he ceased to be a teacher as he became a *text* -- D. J. Boorstin>

Our Display Data type semantically remains to be *text* in the sense of Webster’s definitions **5 b** and **8**. Clearly, word processor documents can contain images such as drawings or photographs. Modern documents can embed video sequences and animations as well. Dictation (audio) is the most important form of pre-written medical narratives. A scanned image of old medical records or of handwriting is certainly text. In this sense, almost everything can be text, which is also supported by the phenomenologic analysis [p. 7] given in the introduction.

2.1.5 Pulling the pieces together

In the previous exploration of the field of text, we separated out the difference between string data elements, where the raw information of characters is sufficient and "display data," where there is use for formatting the text and augment or even replace the text with multimedia information. This means that there will be a character string data type [p. 38] , and a display data [p. 46] type that covers character-based free text and multimedial data.

2.2 Character String

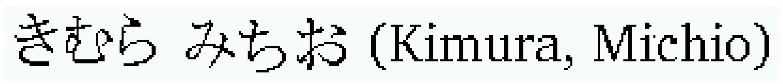
The character string data type for HL7 is a primitive data type. We will not define any data type for the character itself because there is hardly any use for single characters in medical informatics. Therefore a character string is a primitive data type in HL7. Just as it always used to be.

Character String (ST)
A string of characters where every character used by any language anywhere in the world is represented by one uniquely identifiable entity within the string. This type is used when the appearance of text does not bear meaning, which is true for formalized text and all kinds of names.
PRIMITIVE TYPE

Too meet the requirements of international HL7 and globalization of the health care industry, the new data type Character String is developed with this design goal:

> A character string is a sequence of entities each of which uniquely identifies one character from the joint set of all characters used by any language anywhere in the world, now and forever.

For example, one should be able to send Michio Kimura’s (chair of HL7 Japan) name in Japanese Hiragana script and Latin script as


きむら みちお (Kimura, Michio) ,

a string of 24 uniquely identified characters without any switching of character sets.

2.2.1 The Unicode

The Unicode (http://www.unicode.org/) is a character code developed and maintained by an international consortium. The Unicode contains characters of virtually all contemporary scripts, and assigns a unique code to each one of them. Every character in the Unicode is called a "code point". All contemporary scripts fit into the first 65,000 code points. Thus every character can be represented by a 16 bit number.

For example, the string displayed above, would be represented by the following sequence of code points:

```
U+307F, U+3061, U+3049, U+0020, U+304D, U+3080, U+3089, U+0020,
U+0028, U+004B, U+0069, U+006d, U+0075, U+0072, U+0061, U+002c,
U+0020, U+004d, U+0069, U+0063, U+0068, U+0069, U+006f, U+0029
```

Unicode code points are usually written with a leading "U+" followed by 4 hexadecimal digits.

16 bits, i.e., 65536 character code points are enough to accommodate the scripts of all contemporary languages including Latin, Greek, Cyrillic, Armenian, Hebrew, Arabic, Devanagari, Bengali, Gurmukhi, Gujarati, Oriya, Tamil, Telugu, Kannada, Malayalam, Thai, Lao, Georgian, Tibetan, Japanese Kana, the complete set of modern Korean Hangul, and a unified set of Chinese/Japanese/Korean (CJK) ideographs. More scripts and characters are continuously added.

The unified Chinese/Japanese/Korean (CJK) set of ideographs (also called "Han") uses up more than 20000 character positions which is still less than half of the available positions. Acknowledgments should go to those three peoples of China, Japan and Korea, who made a considerable effort of joint standardization work. Given the historical and political problems in this important corner of the world, this is an almost invaluable achievement. If CJK would not exist, we had to reserve for 60000 ideographs!

As the Unicode will expand its scope further into historical scripts (Egyptian or Sumerian) and into such curiosities like the Klingon alphabet, the code would claim another 16 more bits. Since Sumerian and Klingonian languages will not have to be supported by HL7 for even the widest foreseeable future, one can safely assume that every character can be represented in 16 bits.

2.2.2 No Escape Sequences

The most important practical difference to the old v2.x ST data type is that, on the application layer, escape sequences are no longer defined. This is a great relief for application programmers and it will reduce many interfacing problems.

In the example sequence of Unicode characters above, one can look at any position in the string and find a character without having to keep track of escape sequences that switch character sets. For example, we can randomly pick the 5th character from the string, which is U+304D, a Hiragana "mi". The 20th character is a Latin "c". We can tell this without having to watch out for character set switching escape sequences.

Again, there will be no escape sequences defined for the character string data type on the application layer, not for switching of character sets, nor for any other purpose. Notably, the application layer has no idea about "delimiter characters", used by some Implementable Technology Specifications (ITS). To be ignorant of delimiters is a *requirement* if HL7 is going to support multiple ITSs (e.g., for XML, for CORBA, etc.)

This strong position will greatly improve robustness of HL7 interfaces, since application programmers need not worry about whether some characters in strings might collide with a delimiter used by some ITS. The application can use verticle bars "|", carats "^", ampersands "&", less-than "<", quotes """, or any other character. No interference with the underlying message encoding will happen.

2.2.3 ITS Responsibilities

The Unicode gives us a model and the tools to cleanly distinguish between the character string on the application layer and the bytes on the transport layer. This data type specification is focused on the application layer independent from underlying transport. On the application layer, characters are characters, not bytes.

Ultimately those characters must be turned to bits sent over a "wire" to another application. On the transport layer, the Unicode characters can be encoded in different ways. Most straight forward, each Unicode character in the string is represented by two consecutive bytes. But there are other encodings for Unicode characters defined by the Unicode, called *Unicode Transfer Format* (UTF). Major transfer formats are:

- UTF-16, where every Unicode character is represented by at least 16 bits. This encoding is extensible so that 32 bit Unicode characters can be accommodated as well.

- UTF-8, [cf. RFC 2279 (ftp://ftp.isi.edu/in-notes/rfc2279.txt)], where every Unicode character is represented by a variable number of bytes ranging from 1 to 6 bytes, and capable of encoding up to 32 bit Unicode characters. Notably, UTF-8 is backwards compatible to 7 bit US-ASCII, which means that 7 bit US-ASCII strings that do not contain special escape sequences, are UTF-8 compliant. UTF-8 uses the highest bit to signal multi-byte sequences, and thus requires 8 bit clean transport layers.

- UTF-7, [cf. RFC 2152 (ftp://ftp.isi.edu/in-notes/rfc2152.txt)], is an encoding that uses only seven bit on the transport layer. Like UTF-8, UTF-7 is backwards compatible to US-ASCII, with the exception of the plus sign "`+`" used to signal escape sequences consisting of base64 encoded multi-byte Unicode characters.

Underneath the application layer specification of HL7 there is an Implementable Technology Specification (ITS). The task of encoding Unicode characters for transport through the wire is, by and large, assigned to the ITS. The software components implementing a certain ITS must translate characters from and to bytes using some encoding scheme, such as UTF-8.

HL7 interface toolkits that implement ITSs should deal with uniquely identified character entities on the application programming interface (API) side and should always produce proper encodings on the HL7 wire. Applications that would use such an HL7 interface toolkit should have no obligation to deal with character set switching escape sequences or escaping of characters that might interfere with the ITS.

2.2.4 HL7 Applications are "Black Boxes"

HL7 and this data type specification continues to make no assumptions on the internal working of HL7 applications. Although we make recommendations that will help implement the standard, HL7 does not specify the internal working of an HL7 application. A particular implementation may violate all the rules of distinguishing application layer and transport layer. Applications may treat character strings as arrays of bytes, if they so choose, *as long as this practice does not lead to a different behavior of the HL7 interface*.

If application designers decide to deal with lower layer issues like character representation on their application layer, they can do so by selecting an ITS implementation that does not do the mapping to and from uniquely identifiable character entities for them. Those application would be HL7 compliant, *as long as they do not behave differently on the HL7 wire*.

For example, a system *SICK-TOS* was written 40 years ago as a monolithic PDP-11 assembler program. If this program behaves according to the HL7 specification, it would be HL7 conformant. On the other hand, a hyper-modern system *SANI-NET* would not be compliant with HL7, if it fails every time it receives an ampersand "&" character in a message element instance of type character string.

This is more important than it may seem: Suppose the system *SANI-NET* would "support" two HL7 ITS interfaces, for XML and for CORBA. If it would receive "&" with CORBA, it should emit "`&amp;`" on the XML wire. And if it receives "`&amp;`" on the XML wire it should emit "&" on the CORBA wire. The easiest way to be HL7 compliant is through separation of the application layer and the ITS layer through an application programming interface (API).

Again, HL7 does not specify the internal working of applications. Thus, the specification must treat any application as a black box. The only issue, the specification may be concerned about is what happens on the HL7 wire. Thus, this data type specification does not even mandate the use of the Unicode. It does not look at how the strings are represented in the application program. All it cares about is described by the following scenario:

Let system *S* send a message *M* to system *T*. That message *M* contains a character string *C* at a data element, for which *T* promises it will store this data element unmodified and will report this data element back later. Now, system *T* sends that message *M*' back to system *S* containing that data element as a character string *C*'. Back at system *S* the character strings *C* and *C*' must be exactly equal. That is, every character c_i at position *i* in *C* must be the same character as c_i' at the same position *i* in *C*'.

A more concrete example. Suppose your system promises to store a data element of type character string and to report that same data element back to me later. My system uses Unicode characters internally, and I send to you a Devanagari OM character (U+0950)

ॐ

This character would be encoded and sent to your system my means specified in an ITS. Your system receives that message and does with that message whatever it chooses to do. My system does not care what your system does internally, and the HL7 specification does not care what your system does internally. All the HL7 specification claims is that when your system sends that information back to my system, my system should see the same Devanagari OM character (U+0950) on its application layer.

If my system does get back something else, then either my system's ITS layer implementation is broken or your system is broken. This is an operationalization for HL7 conformance on character strings. This type specification demands nothing else.

2.2.5 No Penalty for Legacy Systems

We do not require any application to use Unicode characters internally. And, of course, we can not require that every HL7 conformant application would have to be able to display Kanji, Devanagari or Thai on their user screens.

Applications that can replay any character of the Unicode can be called "high fidelity" applications. But this specification does not even require every application to be high fidelity. For instance, your application could choose to transform any German umlaut "Ä", "Ö", "Ü" to "Ae", "Oe", and "Ue", respectively, and would still conform to this data type specification. This specification allows applications to be high fidelity quite easily, without requiring it from every application.

Legacy systems can comply to this specification, can even be "high fidelity", without having to significantly change their software! This is possible through UTF-8 encoding.

Suppose your application handles 8 bit characters internally and only displays US-ASCII characters. Your application would be conformant to HL7 with any ITS that allows the use of UTF-8 encoding. Any data that originates in your system would use only the US-ASCII character set, which automatically conforms to UTF-8. If you receive data originating from other applications, and if that data contains Unicode characters beyond US-ASCII, your application will not be able to sensibly display the characters, but it can store the characters in its data base byte by byte. Your application would later send those UTF-8 bytes in HL7 messages, thus it would be a "high fidelity" application.

If your application chooses to transcribe foreign characters to US-ASCII (e.g. German umlauts to "AE", "OE", and "UE", or "Kimura" in Hiragana to "KI-MU-RA"), it could display the character strings on US-ASCII terminals. If it transcribes the characters only for the display purpose, but keeps the original code in its the data base, it would still be a high fidelity application.

If your application transcribes the foreign characters as they come in over the HL7 interface, it would no longer be a high fidelity application, but could still be compliant with this specification, with the restriction that it could not claim "high fidelity". To be high fidelity on characters is not so important for end user systems anyway, but it is quite important for data repositories that are to be marketed or used internationally.

High fidelity is possible if you use an ITS with UTF-8 encoding and

1. your communication is 8 bit clean,
2. your data base storage is 8 bit clean
3. you do not use the 8th bit for string delimiters internally
4. your screens won't garble up when being sent 8 bit UTF-8 encoded sequences.

For example, the Regenstrief Medical Record System (implemented using VAX BASIC) would do fine with criteria 1 and 2. It's problems would be located at 3 and 4, though, since it uses delimiters characters internally that are selected from the code range between 128 and 255. Furthermore, the screens would probably garble up when being sent UTF-8 bytes greater than 128.

In this case, i.e., if your environment is not fully 8 bit clean, you can use UTF-7 encoding instead of UTF-8. UTF-7 has the same backwards compatibility features as UTF-8, but does not use the 8th bit. So you won't have conflicts with your internal use of the 8th bit and your communication can strip off the 8th bit if it wants to.

For Europeans, who used ISO Latin-1, the backwards compatibility issue is not as easy as for systems that used only US-ASCII characters. Even though the Unicode itself is backwards compatible to ISO Latin-1, there is no Unicode transfer encoding that leverages this. In the course of this data type working group, we tried to pursue the Unicode maintainers to adopting a more flexible UTF character encoding that would allow backwards compatibility to Latin-1 and other ISO 8859 character sets. However, we did not succeed, more UTF specifications are not welcome. Notably, it were the European Unicode participants who did not think that such a UTF would be a good idea.

It is the task of the ITS layer software to convert any incoming character encoding into the encoding that the application can handle. There is no requirement for applications to use Unicode internally and no requirement for ITS to not support other character encodings, such as ISO Latin-1, or the various Japanese character encodings. The ITS layer software would translate the characters to any kind of encoding that the application software can handle. For example

- For most Java-based applications the ITS layer would most likely convert incoming UTF-8 byte format to Java Strings, which use 16 bit per character internally. This is a basic functionality of the Java core API.

- Most UNIX-based C and C++ character functions treat one character as an `int` (16 or 32 bits depending on the CPU native word size) not as a byte. However the quick and easy approach in C is to use a `char *` as a string, which is just an array of 8-bit characters.

  For those and many other environments that stick to the equation 1 char = 1 byte, the application could choose to use UTF-8 strings internally where the normal US-ASCII characters are represented as single bytes. Those applications would tell their ITS software that it should convert everything to UTF-8.

- A very old legacy system that internally uses a `packed array of char` where a character has only 7 bits, or that for some other reason strips off the 8th bit, would tell their ITS implementations to convert incoming characters to UTF-7 instead.

The key issue is that the ITS layer always performs some translations on the character encoding according to the encoding of incoming messages and the needs of the application. Although HL7's scope is on the message format only, we do recommend that implementors of ITS layers be aware of this character encoding feature they should implement. What is important is that the notion of different character encodings does not exist on the HL7 application layer. No HL7 specification would be valid that makes any assumptions about character encodings or encoding-related escape sequences on the application layer.

2.2.6 Unicode and XML

Using Unicode with an XML-based ITS is the most natural thing to do, since XML is itself aware of the Unicode and its encodings UTF-8 and UTF-16 are required features of every XML parser. In fact, the XML concept of characters served as a model for this HL7 data type specification. The XML specification (http://www.w3.org/TR/1998/REC-xml-19980210) states:

> 2.2 Characters
>
> A parsed entity contains text, a sequence of characters, which may represent markup or character data. A character is an atomic unit of text as specified by ISO/IEC 10646 [ISO/IEC 10646]. Legal characters are tab, carriage return, line feed, and the legal graphic characters of Unicode and ISO/IEC 10646. [...]
>
> [...]
>
> The mechanism for encoding character code points into bit patterns may vary from entity to entity. All XML processors must accept the UTF-8 and UTF-16 encodings of 10646; the mechanisms for signaling which of the two is in use, or for bringing other encodings into play, are discussed later, in "4.3.3 Character Encoding in Entities".

XML 1.0, 2.2 Characters (http://www.w3.org/TR/1998/REC-xml-19980210#charsets)

Since XML uses Unicode internally, there is no need and no way to specify different character encodings in different sections of an XML based HL7 message. There is no interference of Unicode and XML whatsoever. Thus the requirements to character strings stated here are no obstacle to using XML.

2.3 Display Data

To cope with the various encoding formats of appearance, there will be only one data type for both character-based free text and multimedia data. This type is called "Display Data" and will have essentially two semantic components: It will (1) contain the data component and (2) specify the application which can render that data. The application to render the data will be specified by a media type code, similar to the Internet MIME standard [cf. RFC 2046 (ftp://ftp.isi.edu/in-notes/rfc2046.txt)] or HL7 v2.3’s ED data type. The only problem is what data type to use for the data component.

Some formatted text could be defined on top of string data. Due to the backwards compatibility of Unicode to ASCII and ISO Latin-1, the simple typewriter-style formatting, the TROFF escape sequences that were used by HL7’s old data type FT and HTML/SGML formatting is possible on top of Unicode strings. In addition to the string data, we have to indicate the formatting method that should be used by the receiver to render a given string correctly.

Most proprietory text formatting tools, however, do not fit in the character string, because those application use their own proprietory byte encoding of characters and their display properties. Proprietory word processor files and multimedia data is best regarded as an opaque sequence of bits (or bytes) that are rendered by a special application software that understands the given stream of bits. For those, we need to go back behind the character strings to raw bits and bytes.

There seem to be two options. Either we consider it be the task of the ITS layer (the encoding rules) to support the communication of raw bytes data, or we encode raw bytes in strings using the base64 encoding.

With the traditional HL7 encoding rules that were unable to encode raw bytes, raw data had to be sent on top of character strings. This, however, is wasteful for encoding rules and transport channels that can send and receive raw bytes easily. In our definition of a Character String [p. 38] it is wasteful to first construct character strings from bytes, only to transform the character strings back to bytes.

It therefore seem reasonable to define a data type for raw byte strings to complement the character string data type. The raw byte type would be used only by the Display Data [p. 46] type, though. There is hardly any use case for HL7 application domain Technical Committees to use byte string data types directly.

Using byte strings instead of character strings for display data is not only a good idea for proprietory application data or multimedia data, but is also supported by a closer look to standards such as HTML, SGML or TROFF. While those formats are defined on a notion of characters instead of bytes, the applications that implement HTML, SGML or TROFF, have their own means to interpret byte streams as character encodings (e.g. HTML has a `META` element and XML defines the character set in its `<?XML encoding=...?>` processing instruction element. More traditional formatting with TROFF is not even able to handle the full abstraction of characters that comes with Unicode and thus is also based on byte strings rather than character strings.

As a conclusion, we can uniformly define the display data type as the pair of media type selector and raw byte data. If the sender does not want to use any of the format options for display data but just wants to send the raw characters, he can indicate this with a special media type (`text/plain`). Since the display data type is most commonly used for character-based free text, the plain text media type is the default.

2.3.1 Display Data

Editorial Note: In previous releases of this draft specification this data type was called "Multimedia Enabled Free Text" or "Free Text" and was abbreviated "FTX." The name change to "Display Data" was strongly suggested because of considerable confusion caused by term "text" applied to multimedia data. In spite of the drastical name change the functionality of this data type has not changed at all.

The display data type supports both character-based free text and multimedia data and consists of the following components:

Display Data (DD)			
The display data type can convey any data that is primarily meant to be shown to human beings for interpretation. Display data can be character-based free text, whether unformatted or formatted, as well as all kinds of multimedia data.			
component name	**type/domain**	**optionality**	**description**
media descriptor	Code Value [p. 58] using IANA defined MIME type codes	**optional** defaults to `text/plain`	used to select an appropriate method to render the display data
data	Binary Data [p. 52]	**required**	contains the display data as raw bytes
compression	Code Value [p. 58] IANA defined code	**optional**	indicates that the raw byte data is compressed and what compression algorithm was used
charset	Code Value [p. 58] IANA defined code	**optional** for character-oriented media types defaults to the encoding used for Character String [p. 38]	in case of character based media, indicates the character set/encoding of the raw byte data
...			

Other components may be defined for certain media types. This serves as a way to map MIME media type "parameters" to this Display Data type. An example is the *charset* component, which is a parameter of the MIME media type `text/plain`.

The media type descriptor of MIME RFC 2046 (ftp://ftp.isi.edu/in-notes/rfc2046.txt) consists of two parts:

1. the "top level media type", and
2. the media subtype.

However, this data type specification treats the entire media type descriptor as one atomic Code Value [p. 58] .

MIME media types and subtypes are defined by the *Internet Assigned Numbers Authority* (IANA). Currently defined media types are registered in a data base (http://www.isi.edu/in-notes/iana/assignments/media-types/) maintained by IANA. Any of the IANA defined media types is in principle allowed for use with the Display Data type. But not all media types have the same status in this specification.

The following top level media types are currently defined by the IANA:

NAME	PURPOSE
`text`	written textual information
`image`	image data
`audio`	audio data
`video`	video data
`application`	some other kind of data
`multipart`	data consisting of multiple MIME entities
`message`	an encapsulated message
`model`	"an electronically exchangeable behavioral or physical representation within a given domain" [RFC 2077 (ftp://ftp.isi.edu/in-notes/rfc2077.txt)]

There are currently more than 160 different MIME media subtypes defined with the list growing quite fast. It makes no sense to list them all here. In general, all those types defined by the IANA may be used. The downside is that so many options may lead to interoperability problems.

Therefore, this specification prefers certain media types over others and thus assures that there is a greatest common denominator on which interoperability is not only possible, but that is powerful enough to support even advanced multimedial communication needs.

Any IANA defined media type is classified as one of the following for categories:

mandatory

Every HL7 application must support at least the mandatory media types if it supports a given kind of media. There should be one mandatory media type for each kind of media (e.g. written text, image, audio, video, etc.). Without a very minimal greatest common denominator we cannot guarantee interoperability. The set of mandatory media types, however, is very small so that no undue requirements are forced on HL7 applications, especially legacy systems.

In general, no HL7 application would be forced to support any given kind of media other than written text. For example, many systems just do not want to receive audio data, because those systems can only show written text to their users. It is a matter of application conformance statements to say "I will not handle audio". Only if a system claims to handle audio media, it must support the mandatory media type for audio.

recommended

Other media types are recommended for a particular purpose. For any given purpose there should be only very few additionally recommended media types and the rationale, conditions and assumptions of such recommendations must be made very clear.

other

By default, any media type falls into the category *other*. This category means, HL7 does neither forbid nor endorse the use of this media type. Given that there will be a mandatory or recommended type for most practically relevant use cases, the other media types should be used very conservatively.

deprecated

Some media types are inherently flawed, because there are better alternatives or because of certain risks. Such risks could be security risks, for example, the risk that such a media type could spread computer viruses. If a media type is classified as *deprecated*, the rationale must be stated and equally viable alternatives suggested. Not every flawed media type is marked as deprecated, though. A media type that is not mentioned, and thus considered *other* by default, may well be flawed.

The following list shows the categorization of media types according to the above mentioned rules.

Categorization of Important Media Types

MEDIA TYPE	CATEGORY	USE CASE
Text		

`text/plain`	**mandatory** default	for any plain text. This is our former TX data type.
`text/x-hl7-ft`	**recommended** for compatibility to HL7 v2.x	this represents the old FT data type. It’s use is recommended *only* for backwards compatibility with HL7 v2.x systems.
`text/html`	**recommended** could become mandatory in the future	for any marked-up text, sufficient for most textual reports, platform independent and widely deployed.
`application/pdf`	**recommended**	for written text as completely laid out read-only documents. PDF is a platform independent, widely deployed, and open specification with freely available rendering tools.
`text/sgml` `text/xml`	**recommended** for PRA documents	There is a risk that general SGML/XML is too powerful to allow a sharing of general SGML/XML documents between different applications. However, this media type is to be used to convey documents conforming to the HL7 Patient Record Architecture.
`text/rtf`	**other**	this format is widely used, but it has its compatibility problems, it is quite dependent on the word processor, but may be useful if word processor edit-able text should be shared.
`application/msword`	**deprecated**	this format is very prone to compatibility problems. If sharing of edit-able text is required, `text/plain`, `text/html` or `text/rtf` should be used instead.
	Audio	
`audio/basic`	**mandatory**	this is the absolute minimum that should be supported for any system claiming to be audio capable. The content of the "audio/basic" subtype is single channel audio encoded using 8bit ISDN mu-law [PCM] at a sample rate of 8000 Hz. This format is standardized by: CCITT, Fascicle III.4 - Recommendation G.711. *Pulse Code Modulation (PCM) of Voice Frequencies*. Geneva, 1972.
`audio/k32adpcm`	**recommended** for compression	this allows compressing audio data. It is an Internet standard specification [RFC 2421 (ftp://ftp.isi.edu/in-notes/rfc2421.txt)]. its implementation base is unclear.
	Image	

`image/png`	**mandatory**	*portable network graphics* PNG (http://www.cdrom.com/pub/png/) a widely supported lossless image compression standard with open source code available.
`image/gif`	**other**	GIF is a nice format that is supported by almost everyone. But it is patented, and the patent holder, Compuserve, has initiated nasty lawsuits in the past [The GIF Controversy: A Software Developer's Perspective (http://www.cloanto.com/users/mcb/19950127giflzw.html)]. No use to discourage this format, but we can not raise an encumbered format to a mandatory status.
`image/jpeg`	**mandatory** for high color images	This format is required for high compression of high color photographs. It is a "lossy" compression, but the difference is almost unnoticeable to the human vision.
`image/g3fax`	**recommended** for FAX	this is recommended only for fax applications. The format is not well compressed and G3 software is not very widespread.
`image/tiff`	**other**	although TIFF (Tag Image File Format) is an international standard it has a lot of interoperability problems in practice. Too many different versions that are not handled by all software alike.
`image/x-DICOM`	**other**	not sure whether there is an interoperable image file format in DICOM. I know of Papyrus, but is it a DICOM standard?
	Video	
`video/mpeg`	**mandatory**	this is an international standard, widely deployed, highly efficient for high color video; open source code exists; highly interoperable.
`video/x-avi`	**deprecated**	the AVI file format is just a wrapper for many different "codecs"; it is a source of lots of interoperability problems.
	Other	
`model/vrml`	**recommended**	this is an openly standardized format for 3D models that can be useful for virtual reality type of applications and is used in biochemical research (visualization of the steric structure of macromolecules)
`multipart`	**deprecated**	This major media type depends on the MIME standard, the Display Data type uses only MIME multimedia type definitions, not the MIME message format

`message`	**deprecated**	This major media type this is used to encapsulate e-mail messages in delivery reports and e-mail gateways, not needed for HL7. HL7 is itself a messaging standard that defines its own means of delivery and HL7 is not used for e-mail.

Constraints may be applied on the media types whenever a Display Data type is used, whether at the time of HL7 message specification, or for a given application conformance statement, and even in the RIM. For instance, suppose the Image Management SIG will eventually define a class "`Image`". This class Image would conceivably contain an attribute, "`image_data`", declared as Display Data. The IMSIG certainly would not want to see written text or audio here, but only images (and maybe a video clip of a coronary angiography.)

2.3.2 Binary Data

Binary Data (BIN)
Binary data is a sequence of uninterpreted raw bytes (8 bit sequences, or octets).
PRIMITIVE TYPE

The *data* component of the Display Data type is *not* a character string but a block of raw bits. ASN.1 calls this an "octet-string," which is the same as a "byte-string." The important point is that the byte string would *not* be subject to interpretation as characters, but must be passed through from one application’s memory into the other application’s memory unchanged.

The ITS layer has therefore an additional tasks: to facilitate transport of raw byte strings. Transporting bytes is different from transporting characters, this can not be overemphasized. Traditionally, HL7 v2.x roughly supported binary data on top of character string data, either through hexadecimal digits in escape sequence, or through base64 encoding used in the old HD data type. However, this makes only sense for character-based encoding rules such as the traditional HL7 encoding rules or XML. An efficient CORBA ITS, would not need this, as CORBA allows you to transfer raw bytes without trouble.

> Just as character encoding is an ITS layer issue, the encoding of bytes is an ITS layer issue too. On the HL7 application layer we do care only for the unchanged communication of a byte string.

However, when the multimedia type is used to convey plain text, the binary data will be ultimately interpreted as plain text. Through this, character encoding should not be sneaked into the application layer.

The ITS layer software should discover the special case of `text/plain` media and perform the character set translation according to the character encoding used for ordinary Character String data. The ITS layer software can reuse the same machinery that handles character string encoding.

If for any reason the plain text data is in an encoding different from the character encoding used by character strings, this can be indicated through the *charset* component. The IANA maintains a code of character sets (http://www.isi.edu/in-notes/iana/assignments/character-sets) that must be used for this purpose. This IANA code mentions many synonyms for each encoding. If one of them are identified as the "preferred MIME name" it must be used instead of the other synonyms. If none of them is defined as preferred by IANA, the first name mentioned should be used.

With `text/plain` we have the issue of how lines are terminated. The termination of lines must be standardized. The proper interpretation of the ASCII and Unicode standard suggest that line terminators consist of the two control characters *carriage return* U+000D and *line feed* U+000A. This is also the Internet standard of terminating lines and it is native line termination of on MS-DOS descendents. It is easy to comply to those canonical line terminators on Unix systems, who natively use a single *line feed* as an end of line. Apple Macintosh systems use those control characters in reverse order, and those must be swapped.

It is often useful to compress binary data, e.g. using the "deflate" byte stream compression algorithm. This is used by gzip, and pkzip. Almost all data can be subject to byte stream compression (except GIF, JPEG and MPEG, which are already maximally compressed.) Using a media type of `application/gzip` for compressed data is obviously not useful, since it would override the description of the uncompressed data. The component *compression* is to be used instead. Either an IANA code is to be used or a subsequent revision of this specification will mention a table of allowed codes.

2.3.3 Outstanding Issues

We will define a code for compression algorithms.

We recognized that there will be a reference data type defined to be used alternatively for huge data blocks. Should the Display Data type be allowed to be replaced by a reference, or should it contain a reference?

Video streams do not fit into a single message, an external stream protocol (such as RealVideo) would be used. This could be accommodated through a reference data type.

3 Things, Concepts, and Qualities

3.1 Overview of the Problem Space

3.1.1 Concept vs. Instance

Most medical information comes as qualitative information: complaints, symptoms, signs, diagnoses, goals, interventions, surgeries or medications, all of these are informations on a nominal scale. But not only medical information, administrative data often is on nominal scales too, e.g., patient class (inpatient, outpatient, etc.), insurance, health plan, and many other data elements. These nominal scaled values are variables that can take on one value of a list of possible values.

The semantic field on which we are now focusing contains more than just values on nominal scales. Values on nominal scales are abstract concepts. For instance, the color *green* is such an abstract concept. There is not some tangible *green* anywhere in the world, only bodies whose color is green, or green light being reflected from bodies (which is the same physical phenomenon.) Likewise, there is no *pneumonia* to which we can point and say: "here comes Pneumonia!" And although we would say that a Streptococcus pneumoniae bacterium is a real physical body, we usually are not interested in the one bacterium lying in the lower left corner of our microscope view area. What we are interested in is the concept of Streptococcus pneumoniae, not the individual bacterium.

On the other hand, we often need to refer to individual things, like *this* table, or *this* computer on which I type. Individual things can be classified into concepts, *a* table or *a* computer. But when we want to refer to individual things, we just do not want to classify. Referring to individual things is thus the opposite from referring to concepts. In our data type model we have to serve both needs, referring to concepts and referring to individual things. We can call individual things "instances".

However, the distinction between "concept" and "instance" is not very crisp. Philosophically we can easily argue that Gunther Schadow is a mere concept, (you might have seen me, but that is not essential for your concept of Gunther Schadow). Through my writing I am currently a concept in your mind that might have more or less shape but still it is likely to exist only in your mind. Although Julius Caesar or Napoleon may have been real existing creatures, they now persist as mere concepts.

An instance is something you can (merely) point to or touch or destroy. A concept can not be pointed at, touched or destroyed. A concept can only be explained. Both, instances and concepts have names, although these names have different characteristics. I, as a living human being, am an instance and I have a name: "Gunther Schadow." By contrast, "headache" is something one can explain. When you feel you own headache, your present headache might even become an instance for you, but your particular headache is rarely an instance for others.

Thus "headache" is a concept. The Hypertext Transfer Protocol (HTTP) is a concept as well. You can not point at HTTP, you can not touch HTTP, you can not destroy HTTP. But you can explain HTTP. You can explain HTTP to your wife, but you can not explain Gunther Schadow to her. You can tell her about your experience with meeting me on phone or e-mail, but you can not "define" or explain Gunther Schadow. Instances can be assigned to categories. You can say that Gunther Schadow is a human, male, and living in Indianapolis. That categorizes me in certain manners, but it does not explain me.

Generalization and specialization are relationships between concepts, not instances. Gunther Schadow does have neither a generalization nor specializations. We too frequently blur that distinction between concepts and instances, when we talk about a "parent-class" or "children of a class." Parent/child are relationships between instances, not classes. But the metaphor of the genealogy for looking at concept-relationships is very very old (Porphyrius, an early commentator, perhaps a student, of Aristoteles.)

Gunther Schadow has parents and I do have a child. Headache has no parents and "tension headache" may be a specialization of headache, but it is never a child of headache. As such the very term "inheritance" is distracting, since inheritance exists only among instances, not concepts. We have to be very careful about our metaphors.

3.1.2 Real World vs. Artificial Technical World

The term "instance" is also in opposition to "class". In the object-oriented paradigm (actually originating with Aristoteles 400 b.C.) there are classes that are concepts of real things and instances (or individuals) which are the real things themselves. In object-oriented language we would probably want to say "class" vs. "object", however, this distinction is ambiguous, since people often point onto a box in the RIM labeled "Patient" and say "this is the Patient object". It is the class, not the object. But of course, when dealing with classes in computer systems, they too become objects (sometimes called meta-objects).

Within computer systems everything tends to blur up. Every object oriented language has pointers (or references) to objects (= instances). Some treat classes as meta-objects (e.g., Java does). In any case, an instance in your computer memory or on a file can be pointed to (using an index, pointer, database key, or whatever). It can be "touched" (modified, directly examined), it can well be destroyed. But it can not be explained. It can be copied, though, and as such it becomes like a concept. But "real" object-oriented systems (like CORBA) do not allow you to just copy an instance.

Computer systems shed a whole new light on the problem space. There are class instances on healthcare information systems, that refer to some real world instances. Thus, a record in a patient registry refers to a real existing patient. Both the patient record and the human patient are related, but not the same. Thus there is a new pair of antonyms: real things and reflexions of things within information systems.

Although, HL7 deals primarily with reflexions of things within computer systems, there is this important linkage between the information about things and the real things themselves that HL7 must care about.

It is very difficult to unambiguously link to real things. This is because instances can only be pointed at. I can say "this table", but "this" does not mean anything for you if you are not here in my office. I can describe my desk to you, but you will not be able to recognize the individual desk from among others of the same kind. The only thing one can do about this is to search for individual properties that only my desk has, e.g., a particular scratch. Thus, we can collect information about instances and use this information to refer to the instance, in the hope that there will be no second instance that matches the same description. But you never know.

An alternative to describing the scratches at my table is to put a tag on it with an inventory number. My computer screen, for example, has such a tag on it with the inventory number "2464" assigned by the Regenstrief Institute. Inventory numbers are a common way to refer to individual things, we can easily put a tag on them.

But we can not put a tag on people. There is a custom to brand animals, but luckily in our culture we do not brand human beings. We give names to human beings, names they remember from about their second year after being born and until several days before dying, at average. However, names do change, names are misspelled. Everyone who deals with healthcare informatics knows something about the problem to identify people.

On the one hand, with computers and technical devices, some things become easier. For example, real world concepts such as diseases or even colors are hard to describe. Modern science tries to operationalize concepts, i.e. to provide a protocol by which you can reproduce an instance of that concept or by which you can decide whether something is an instance of a given concept or not. But operationalizations are a matter of consensus, and that consensus does often not exist, not in everyday life and not in the sciences. Conversely, with computers and technical devices concepts have crystal clear definitions and instances have exact locations and extent. For example, HTTP has a specification that tells you exactly what to do to become HTTP compliant, and that allows you to exactly decide whether or not you deal with an HTTP interface. If I dial a telephone number, there will be precisely one phone ringing somewhere in the world.

3.1.3 Segmentation of the Semantic Field

In the introductory approach to our semantic field we found two pairs of terms that seem to cover a lot of the phenomena that we have discovered: concept vs. instance and real-world vs. technical artifacts. We try to sort out the phenomena we have to deal with in HL7 with the following 2x2 matrix.

	CONCEPT	INSTANCE
REAL WORLD	Coded using mostly externally defined code systems: ICD9, ICD10, SNOMED, DSM-III, DSM-IV, ICPC, LOINC, ICPM, CPT4, etc.	Examples: person names (old PN), organization names (old XON), locations descriptors (old AD, and PL), legal id numbers (SSN, DLN, etc.)
TECHNICAL	Examples: message type, order status code, participation type code, MIME media type.	Examples: message ids, Service catalog items, RIM instances (order numbers), phone numbers, e-mail addresses, URLs

REAL-WORLD CONCEPTS are concepts that scientists and ordinary people deal with in their minds and formulate in words. Communication must rely on commonly agreed terminology or standard code systems. Those are mostly defined by external (i.e. non-HL7) organizations, such as those organizations representing domain experts in a particular medical specialty.

There is currently a lot of overlapping, competition and complementation of code systems. It does not seem as if this apparent disorganization could ever change because medicine and human life in the real world is always changing. Thus, the communication of real world concepts will always have to deal with issues of translating codes selecting the best matching "synonymous" code from different code systems.

TECHNICAL CONCEPTS are labels for well-defined concepts, such as protocols. For example: if we say "HTTP" we refer to the hypertext transfer protocol, that is an Internet standard defined quite rigorously. If we ultimately want to know what HTTP is, we can read the specification. However, most often we are not so much interested in what "HTTP" *is* or in what *its meaning* is, but we just want to use it. So we select an appropriate machinery (i.e. a web browser) and use HTTP.

With *Technical Concepts* there is no use for different vocabulary, no use for using both "HTTP" and "HypTexTranProt" to refer to the same technical concept. This is not to say that people could not use different names or abbreviations for HTTP, but it means that there is no point in letting everyone choose his own terminology for the exact same technical concepts.

REAL WORLD INSTANCES are individual people, organizations or things that we can meet, point at, think of, go to, etc. The strongest "definition" we can ever make is to point at those people or things, touch them or take them into hands and show them. But in documents and human communication we commonly use Names, some officially assigned Identifiers (i.e. social security number, or driver license number). Places are named using residential addresses, or other kinds of locators (e.g., building->tract,->floor->room->bed).

Things are most often pointed to (e.g. "give me *this* screwdriver"), or described (e.g., "give me the long screw driver ... no, the stronger one"). In larger context where we can neither point to things, nor could unambiguously describe things, we just assign arbitrary inventory numbers to the things.

In general, identifiers for *Real World Instances* are quite rich of intricacies and we will address those later. The common approach for data types is already laid out by HL7 v2.x: i.e. PN, XON, DLN, AD, PL, and the like.

TECHNICAL INSTANCES are instances that are useful in some technical sense. Just like with *Technical Concepts* we are less interested to *know* what exactly those instances are. Rather, the reason why we name technical instances is because we want to *use* them. In case of HL7 most of those technical instances will be particular data instances, such as messages, order numbers, service catalog items, or any other instance of a RIM class that we can refer to.

But *Technical Instances* are also things like telephone numbers and e-mail addresses or *Uniform Resource Locators* (URL) to Web pages, images, or chat rooms. The general idea is that what you do with a phone number is to pick up your phone and call your party. You would not search the phone book in order to find the address of where a given telephone is located and to meat your party there. Searching the phone book for an address would be to find out what a given telephone number *means*. In most cases, we choose to directly *use* those telephone numbers by simply picking up the next phone and dial that number.

The same is true for database records or data instances on computer systems, we do not go and analyze memory dumps of computer systems in order to find out what a given *Technical Instance* really is, we just use them in some machinery that, for instance, lets us query for a given record entry, lets us change that record entry.

3.2 Technical Concepts and the Code Value

The **Code Value** data type is the basic building block for referring to concepts, both technical and real world concepts. A Code Value is essentially a symbol with all contextual information necessary to interpret that symbol, i.e. the literal and the code system that defines a given literal.

Code Value (CV)			
A code value is exactly one symbol in a code system. The meaning of the symbol is defined exclusively and completely by the code system that the symbol is from.			
component name	**type/domain**	**optionality**	**description**
value	Character String [p. 38]	**required**	this is the plain symbol, like "`784.0`"
code system	*a code by itself*	**required**, can be fixed by context	denotes the code system that defined the plain symbol
code system version	Character String [p. 38]	**optional**	a version descriptor *defined specifically for the given code system.*
print name	Character String [p. 38]	**optional**	a sensible name for the code as a curtesy to an interpreter of the message. **THE PRINTNAME HAS NO MEANING, it can never be sent alone and it can never modify the meaning of the code value**
replacement	Character String [p. 38]	**conditional**, iff *value* is not set	a name for the concept to be used in case that the concept is not codeable in the specified coding system. If the *value* attribute is set, the *replacement* attribute MUST NOT be set. **In no way can a replacement string modify the meaning of the code value**

For example

```
(CodeValue
   :value      "text/html"
   :codeSystem "MIME-TP")
```

would refer to the technical concept "HTML media type", while

```
(CodeValue
   :value      "784.0"
   :codeSystem "ICD9 CM")
```

would refer to the real world concept "headache" as defined by ICD9 CM (i.e., in ICD9 CM, this concept of headache does not include the concept of "tension headache", 307.81).

Technical concepts will be referred to simply by using the Code Value. The Code Value will also be used as the building block for more complex real-world concepts.

The code system is a mandatory component of the Code Value data type. However, in a given message it need not be sent, if it is fixed by the context. For example, in an HL7 message header field designating the event code, only one coding system is allowed, i.e. the HL7 event code. It would only be redundant to send a code system identifier for a code value in that place.

It is recommended that HL7 interface software that knows about the default code system fill in the default code system component before handing the Code Value to the application layer software. The strong binding to the field in the message header may get lost while the message is processed, and thus the default code system may no longer be inferable later.

In fact, an implicit type conversion rule exists between Character String and Code Value. If in a given field is declared as a Code Value with a mandatory code system, but the message contains a Character String in that field, the character string found is taken as the value part of a Code Value and the mandatory code system is taken as the code system identifier. An exception is raised when the supplied character string is not a defined symbol of the mandatory code system.

The above conversion rule allows to build concise messages with code values, just like the HL7 v2.x ID data type allowed one to do.

Outstanding Issues

The `code system` obviously is by itself a *technical concept* identifier. If we are going to use the data type *Code Value* for concept identifiers, we have a recursive type definition. Recursion is not a bad idea in general, but the question is: what terminates the recursion?

If HL7 maintains a list of coding schemes and defines symbols for any one of those schemes, we can circumvent this problem of recursion by defining the component named `code system` as a simple *Character String*. We can continue to use the code system register that was used with HL7 v2.x.

What happens if HL7 outsources its code of coding systems? What happens if there are multiple codes of coding systems (e.g. suppose the CEN coding system registry standard becomes an ISO norm?)

HL7 could for all times maintain its registry of coding systems. And if HL7 will outsource the maintenance of the registry of coding systems in the future, it would always require only one backward compatible registry to be used. If we believe that HL7 will for all times maintain its own registry of coding systems, we could shortcut any recursion and instead use a Character

String.

[An alternative would be to use ISO Object Identifiers as coding system identifiers.]

The `code system version` is used as a refinement of the `code system` descriptor. Logically, any version information is useful only together with the code system identifier.

The hard difference between a code system name and a version is problematic. For instance, the question is, whether "`ICD`" is the code system name and "`9`" or "`10`" is the version? If so, what about the derivatives of ICD-9 (e.g., ICD-9-CM) and ICD-10 (e.g., ICD-9-PCS)? What about the minor versions where a few codes are taken out or brought in every now and then? If we define all coding systems in a special HL7-maintained table, we would not need to use a separate version identifier, because the HL7 code system registry could simply define a new code system symbol for every new major and minor version of every code system.

A possible policy to some of this is: whenever a code system changes in an incompatible way, such as between ICD-9 and ICD-10, there will be a new entry in the HL7 registry and thus a new code system identifier will be created. Different versions would only be used for changes that are compatible.

It would not matter how the other organization calls an update of their coding system. For example, WHO speaks about "*International Classification of Diseases, 9th* ***revision***" but HL7 still considers this another coding system, not just another revision or version of basically the same code system. By contrast, when LOINC updates from revision "1.0j" to "1.0k", HL7 would consider this to be just another version of LOINC, since LOINC revisions are backwards compatible.

How can we assure that the stuff people will put into the version component is standardized and interoperably useful?

HL7 would still have to make sure that the true version identifier of LOINC 1.0j is either of "`1.0J`," "`1.0j`," "`1.0-J`," "`1.0 j`," but not just any of those. While the organization who maintains a code system will have their own version numbering scheme, they will not define unambiguous exact string representations for their revision ids. And HL7 can not expect them to define precise character string representations for their version identifiers. Thus, HL7 has to maintain a list of the version identifiers for each code system, or at least a set of clearly defined rules about how the version identifying string can be inferred from the version id used by the other organization.

Unregistered local coding schemes have been the cause of a lot of trouble in the past. Laboratories whose main concern is not HL7, update their code system ids quite frequently and without caring for backwards compatibility. This places a lot of burden on the shoulders of HL7 communication system managers. This burden would not be easier, but heavier, if every ideolectic coding scheme that changes ever so often would have to be registered with HL7.

The answer could be to say that locally defined coding systems do not have any meaning outside the defining organization. Thus, there is no point in registering anyway. As long as the coding system identifiers do not collide with the HL7 defined code system identifiers, it wouldn’t matter if there are code system name conflicts between different sites for their local code systems.

Traditionally, HL7 defined the letter "`L`" to stand for any local system, or, if more than one local code system exists at a given site, to name those "`99`*zzz*" where *z* would be a digit. We can loosen this constraint a little bit by saying that every code system name starting with "`99`" be local.

3.2.1 State of a State Machine

One particular kind of technical concept identifier will occur very often in HL7 messages: state. Since the HL7 version 3 message design methodology bases the definition of messages on State-Transition models, the communication of state attributes will be standardized and stylized.

The notion of a State of a State-Machine will not be defined here in all detail, instead we refer to the HL7 Message Development Framework, to the Unified Modeling Language Specification, and to a vast amount of literature on that matter. Note that the study of Automata (State-Transition-Models) is one of the oldest areas of Computer Science and a basic part of computer literacy.

Objects have identity and state. Identity is fixed by an identifier attribute of an object (or a reference to an object). An object is in one and only one state at any time. The state is the total of all the current values of attributes and all the current associations to other object. Thus, generally speaking, state is far more than could be represented in one state variable; in other words, the state of an object is everything but its identity.

A State-Transition model often focuses at certain distingushed features of an objects possible states. Thus, in a more narrow sense, state variables explicitly capture those states of an object that are defined in the State-Transition model of a class. Every state of a State-Transition model stands for an entire class of actual states that objects might go through in their life-cycle.

Many of such states defined by a State-Transition model will have certain constraints that constrain the attributes and association that must exist or that may not exist for an object in that defined state.

In the following we will use the term **joint state** to talk about the overall state of an object according to a State-Transition model. Note that at any given time an object is in one and only one joint state, independent of the details of the State-Transition models (e.g., no matter whether there are parallel sub-state-machines, or nested state’s used.)

We will use the term **partial state** to refer to the sub-states that a State-Transition model distinguishes individually. An object can be in multiple partial states at the same time. The total of all *partial states* that are effective for an object at any given time is the *joint state* for that object at that time. Note that, generally speaking, all properties of an object can be considered partial states, however, here we call partial states (proper) only those partial states that are defined in the State-Transition model.

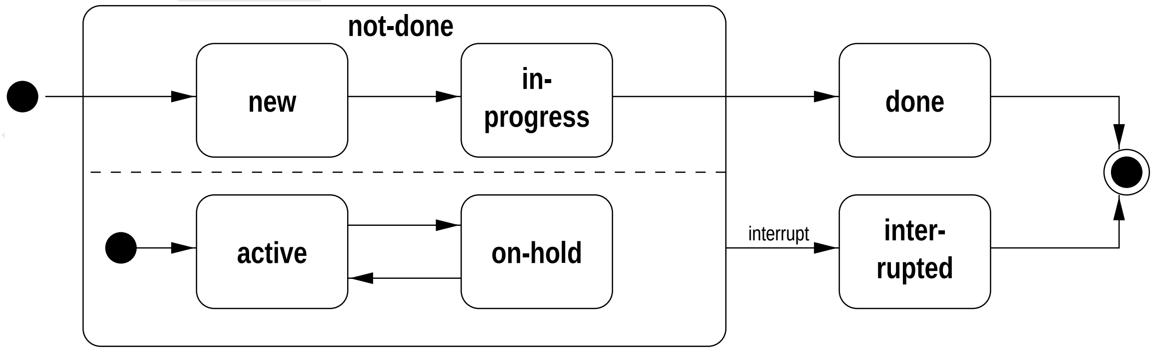


Figure 3: Example State-Transition model.

For a very simple State-Transition model in UML there may be no difference between partial states and joint states. However, in UML concurrent State-Machines partial states are different from joint states. For example, an order may be in the states *new*, *in-progress* and *done*, as shown in Figure 3. At the same time any order may be *active* or *on-hold*. Suppose that transitions to put an order on hold are considered independent from the other three possible states of an order. In that case, the joint state of the order is described by mentioning one partial state of {*new*, *in-progress*, *done*} and one of the states {*active*, *on-hold*}. The set of all possible joint states would be the cartesian product of the two sets of states:

new	*active*
new	*on-hold*
in-progress	*active*
in-progress	*on-hold*
done	*active*
done	*on-hold*

There is another variation of the term "state" distinguished by UML: *composite state* (or *nested state*) vs. *simple state*. Composite states are more coarse-grained states that one may want to distinguish because a transition may be applicable to each of the *component-states* nested within the composite state.

For example, one may want to allow an order in both of the states *new* and *in-progress* to be interrupted. So, one might define another state: *interrupted* and one transition from each of the states *new* and *in-progress*. To express that there is really no difference betweem *new* and *in-progress* for the purpose of interrupting, one can define a super-state, e.g., called *not-done*, to nest both *new* and *in-progress*. Thus, only one "interrupt"-transition would be used from the super-state.

State-Transition diagrams that use nested states are easier to read and comprehend, since they provide abstratctions and generalizations and thus reduce the number of similar transitions. However, the information about super-states does not need to be mentioned explicitly, since it is always implied by its component state. In our example, if either *new* or *in-progress* is effective, we know that the super-state *not-done* is also effective. Thus, explicit information about super-states is always redundant.

Alternatives for designing a data type for state.

ISO 11404 (language-independent data types) defines a data type for state. However ISO defines the state as a simple enumeration of state code. Thus you could only communicate one symbol per joint state in a variable of that type. If you have multiple parallel state machines, in other words, if multiple partial states would be effective at the same time, you would need to precoordinate the list of parallel state codes.

Precoordination of the table of state codes for any given class has its merits. With a precoordinated code, you know that any given value is actually legal. Conversely, for a postcoordination of codes, you do not know whether you have a legal combination unless you explicitly test for it. In our example, in a precoordinated joint state code you were sure noone could utter a state that at the same time includes both *in-progress* and *interrupted*.

Precoordination, however, defers the burden to the time when the information needs to be interpreted. A precoordinated code requires a table that helps to separate the different partial states from the joint state code. Even small changes to the state transition model may entail a number of joint state codes to be added or taken away from the table. On the other hand, if the processing of those state codes were in reality based on a table, there is a lot of built-in flexibility, since a table driven processor should continue to work properly as the driving table is updated. So, a precoordinated state code with one entry per joint state is a good choice.

Obviously the opposite of precoordination is **postcoordination** and thus, we could define the state data type as a vector of partial state code. If the possible partial state codes can be factored into multiple orthogonal axes, it makes sense to label each of the components of that vector of partial states with some descriptive name, in other words, to represent state as **one record of joint states**.

A related alternative to representing the joint state in one attribute of a record type would be to allow the state to be expressed in **multiple attributes**. An example for this is Wayne Tracy's Clinical_document_header class with the four attributes completion status, availability status, authentication status, and storage status. Wayne's approach is currently not conformant with the MDF style, however, Wayne's approach existed before the MDF style and that has the honor of the elder, meaning it can not simply be dismissed as a style guide violation. However, in the following I will stick to the notion of a single state variable per object.

In a postcoordinated code for states the question arises what to do with **composite states**. As noted above, composite states need not be sent in a message since they are always implied by their component states, thus, composite states are, strictly speaking, redundant. However, just as mentioning the generalized composite states in a State-Transition model simplifies definition of the model, having the generalized states on hand might simplify the processing of state information. Indeed, if all a given application is interested in is a super-state to be effective, it is simpler to check for the existence of that super-state flag in a collection of state flags, rather than having to test for every possible sub-state flag.

In our example, the diagram says that the transition "interrupt" is possible from the super-state *not-done* that encloses the sub-states *new* or *in-progress*. It would be convenient for an application to test whether *not-done* is among the set of state flags in the state variable (one test), rather than to test whether the either state *new* or *in-progress* is effective (two tests).

The postcoordinated approach with explicit super-states also simplifies seamless evolution. The following evolutionary developments of State-Transition diagrams are supported:

1. Refinement of a state to include sub-states. This is probably the most likely development. The scenario is that some applications will know earlier than others that the state *not-done* would have turned into a super-state containing *new* and *in-progress*. Since the *not-done* state flag will be continued to be sent in the state variable, old applications continue to work, if they ignore the unknown state flags. Ignoring the unknown state flags is quite natural, since one would rarely iterate over all state flags in the state variable, rather than testing whether particularly known state flags of interest are within the set.

2. "Recoarsement" (antonym of "refinement",) i.e. turning a super-state with sub-states into a state without sub-states. This is probably quite rare. It could occur if a we had an over-design in a State-Transition model, providing features that nobody wants to use and that cause more confusion than benefit. In this scenario, the *not-done* state that had sub-states will turn into a state withgout the sub-states. Since most (if not all) applications in this scenario never asked for the sub-states and only tested for the super-state, they will not even notice that the sub-states are no longer defined in the model.

3. Introduction of a super-state. In our example, suppose our state-transition diagram started without the *not-done* state and two "interrupt" transition were defined from both *new* and *in-progress*. The model would later be simplified to include the state *not-done* with only one transition named "interrupt". Note that the introduction of super-states is a very mild change, and properly designed applications that conformed to the old model will also be conformant to the new model. **However, old applications would not send the super-state flag explicitly in their state variables, which could lead to problems with new applications that do rely on that state-flag to be sent.**

4. Introduction of parallel sub-state-machines. In our example, suppose our State-Transition model did not contain the *active* - *on-hold* sub-state-machine. The introduction of the new parallel states will introduce new state flags in the state variable, but applications that do not depend on those states will just ignore them. In the reverse direction, new applications that do handle the parallel state-machine, need to assume a default state *active* if not otherwise mentioned.

Conversely, the pre-coordinated status code would have changed significantly with every of the above changes and the kind of flexibility we have with the post-coordinated code could be achieved only with an intermediary table for interpretation and mapping between message status codes and application status codes.

I have some UML issues that reinforce me to recommend a little un-dogmatic UML modeling style, which however is not a big difference. In UML a tranbsition from a super-state to one of it's internal sub-states is not defined. Rather UML suggests to use nested initial pseudo-states. However, this requires to explicitly mention both states *active* and *on-hold* which is really redundant. Having both states in the model is redundant because *active* is considered just the negation of the *on-hold* state and does not add any functionality or clarity to the model. The evolution is easier if *on-hold* would just be added as a new feature and the default being automaticly active, if *on-hold* not being mentioned.

Finally another alternative is to use a **post-coordinated state code without mentioning super-states**. On the first glance, the above-mentioned evolution paths rely on the super-state information to be sent. However, one tiny step of indirection in the interpretation of the state variable would open the same evolution path for the minimal set of state flags.

Remember that states are essentially predicates or assertions about objects. The named states, e.g., *new* will be used in predicate statements such as: "`if` state `is` *new* do stuff," or more formally: "`if` *new*(state) do stuff." How would those predicate tests be implemented?

If we had a precoordinated state code, or if we had only one state flag at a time, the program would ask whether the current state equals some state to test for:

```
IF state = new
THEN
  do stuff
ENDIF
```

If you have to test for the state *not-done* if it is not sent explicitly you need to do

```
  IF ( state = new ) OR ( state = in_progress )
  THEN
    do stuff
  ENDIF
```

If *not-done* is sent explicitly, the state variable can not be just one code but a set of state flags. That is, the test would look like

```
  IF not_done IN state
  THEN
    do stuff
  ENDIF
```

if the state variable were a set and super-states, such as *not-done* were not mentioned, you had

```
  IF ( new IN state ) OR ( in_progress IN state )
  THEN
    do stuff
  ENDIF
```

or alternatively (with `*` being the intersection operator)

```
  IF ( { new, in_progress } * state ) <> {}
  THEN
    do stuff
  ENDIF
```

now, even if super-states would not be mentioned explicitly, we could use a table of constants that let the application work the same no matter whether super-states are mentioned explicitly or not:

```
  CONST
    new_mask         := SET { new };
    in_progress_mask := SET { in_progress };
    not_done_mask    := SET { new, in_progress };
```

```
...

IF ( not_done * state ) <> {}
THEN
  do stuff
ENDIF
```

The advantage of this method is that your application code is invariant to whether states are represented explicitly or not. In addition one can test for special state constellations such as *in-progress* AND *on-hold*:

```
CONST
  new_mask         := SET { new };
  in_progress_mask := SET { in_progress };
  not_done_mask    := SET { new, in_progress };
  my_special_mask  := SET { in_progress, on_hold };
...

IF ( my_special_mask * state ) = my_special_mask
THEN
  do stuff
ENDIF
```

As a conclusion, it seems to be very flexible to assume state variable uniformly to be a set of state flags and to test for state flags indirectly through intersections with "mask" sets testing for the non-empty set (OR) or equality with the mask (AND).

In the same way one can conduct checks for the state variable to represent a legal state, e.g., to test for **either** *new* **or** *in progress* to be effective, but not both:

```
CONST
  new_mask         := SET { new };
  in_progress_mask := SET { in_progress };
  not_done_mask    := SET { new, in_progress };

IF CARDINALITY( not_done_mask * state ) > 1
THEN
  THROW Illegal_state_exception;
ENDIF
```

The set operations as shown in the above examples seem to require special programming language support, however, in fact they do not. Sets in Pascal or MODULA 2 are nothing but bit-fields, and the intersection operator is nothing but the bit-AND operation on bit fields. Thus this mechanism is implemented with ease on any programming language such as C, BASIC, you-name-it.

To summarize the above discussion we have found:

1. that a pre-coordinated state code enforces only legal states to be communicated, but interpretation and evolution is difficult and requires a table to interpret and map state codes to something the application can handle;

2. that a redundant post-coordinated state code, that sends super-state information is easy to handle and allows for smooth evolution and interoperability between applications with a different interest in the details of a state-machine;

3. that a post-coordinated state code that does not send super-state information is even more flexible given that state predicates are tested based on state "masks" that can be defined in a simple table.

4. that a pre-coordinated state code will always fit in a single code value;

5. that a post-coordinated state code will rarely fit in a single code value and treating it as a set up-front is a requirement for the discussed evolution rules;

6. that a post-coordinated state code can alternatively be sent in a record of state variables or in multiple state variables, in which case the described flexibility of evolution and interpretation is lost. [There are ways to consolidate multiple state variables in an application, but that is more complex for the sole reason to have multiple state variables in the RIM.]

No decision has been made as of yet. My proposal is to:

1. Define a data type called "State" which makes the actual state representation opaque to the application layer. I don’t want to bother the domain TCs with this "CV or SET" discussion.

2. Stick to the MDF rule of one state variable and try to pursue Wayne that this would work for his part of the standard. However, wait with making the final decision until Wayne has agreed to the harmonization proposal to merge his four state-variables into one. Wayne has the right of the elder here.

3. Use the non-redundant post-coordinated state representation and propose to implementors to test for states uniformly using "masks". Alternatively to go to the redundant post-coordinated alternative, if opposition gets too nervous.

3.3 Real World Concepts

The old CE data type and its interim proposed successors (with various names LCE/CWE and CE/CNE) were basically one pair of Code Value [p. 58] plus a display data string that could be used to convey the original text in an uncoded fashion.

The new data type for real world concepts is essentially a generalization the CE. The Concept Descriptor is defined as a collection of Code Values [p. 58] with one, two, or more codes.

There is an important difference for the semantics of a collection of Code Values [p. 58] . Two those semantic flavors exist:

1. A collection of quasi-synonyms, i.e. codes that have been selected from *different* coding systems in order to convey *the same meaning*.

2. A collection of codes, possibly from the same coding system, that modify the overall meaning.

Both flavors of collections of code values will have to be supported by the new data type for real world concepts. An example from HL7 v2.x is the "specimen source code" in the OBR-Segment, which was such a conglomerate of quasi-synonyms *and* modifiers.

The Concept Descriptor supports the two kinds of collections of Code Values without mixing them all together. The Concept Descriptor data type therefore is a rich nested structure, whose complex structure reflects the complexity of the task it has to perform.

There may be a requirement to the new data type for real world concepts to keep track of the *systems* which perform translations on those codes. Thus, every code value could be annotated by whom, when and how a particular quasi-synonymous code value was added to the collection of quasi-synonyms.

When codes are translated to other codes of other code systems, the original meaning is necessarily distorted. Thus, it does matter which translation occurred based on which prior Code Value. The new data type Concept Descriptor keeps track of the *order* in which translations where performed and on the *quality* of those translations.

The Concept Descriptor [p. 72] is basically a partially ordered set of Code Translations. Every code value is considered one translation. The first code value is the translation from the original text to a code value. Other translations to other code systems may be added to the concept descriptor either based on code values already in the set of translations or from the original text.

Every translation refers to the the translation that it is based on.

Codes and their modifiers are collected in a Code Phrase [p. 74] . The code phrase is an intermediate level between Code Value and Code Translation. That means that every Code Translation contains an entire Code Phrase. Examples are given after the formal definitions of the involved data types.

3.3.1 The Concept Descriptor

The data type for Real World Concepts shall be defined in as the "Concept Descriptor".

Concept Descriptor (CD)			
A concept descriptor communicates a real world concept (such as a finding or a diagnosis). A given concept may be expressed in multiple terms where each term is a translation of some other term, or is a (re-)encoding of the original human readable text.			
component name	**type/domain**	**optionality**	**description**
translations	SET OF Code Translation [p. 73] s	**required**	These are the translations or quasi-synonyms of one real world concept. Every translation in the set is supposed to "say the same thing in different words." The translations in the set form one directed graph that is fully connected.
original text	Display Data [p. 46]	**optional**	This is the original text or phrase entered by a clinician that was the basis for the initial coding. This can also be the text that was displayed to the clinician in a selection menu and thus was the basis for the selection of the particular initial code term in the set of translations.

3.3.2 Code Translation

Code Translation (CDXL)

This data type holds one code phrase as one translation in a set of translations describing a concept. The additional information in this data type points to the source code used in the translation process and describes who or what performed the translation and what the quality of this translation is.

component name	type/domain	optionality	description
term	Code Phrase [p. 74]	**required**	All the meaning of the translation is found here, the rest is descriptive stuff.
origin	reference to CodeTranslation [p. 73]	**required**	This is the code in the list of translations on which this translation was based. This is a required component which means, whoever adds an additional translation must reference the source code. No reference here means that the given translation is the original code.
producer	Technical Instance Identifier [p. 84]	**optional**	This identifier tells what system performed the translation. This information can be useful to audit the translation process or to estimate the quality of the term based on prior experience with the translation of a given producer. This identifier refers to some system, not a particular human coding clerk.
quality	Real Number [0..1]	**optional**	An estimation of the translation quality. This is a value between 0 and 1, where 1 stands for an absolutely accurate translation and 0 stands for random fuzz. We do not require a special method to be used here to estimate the quality. This can just be a subjective estimation of the form we use in eliciting probabilities for a belief network. But we can recommend some example methods of how those values can be computed. We can also map all other quality estimations mentioned in the literature onto the interval [0..1] of real numbers.

3.3.3 Code Phrase

Code Phrase (CDPH)
A code phrase is a list of code values which all together make up a meaning. This can be used for example in SNOMED, where you can combine multiple codes into a new composite meaning. HL7 used to combine codes and modifiers for the OBR *specimen source*. And HCFA procedure codes also come with modifiers.
ORDERED LIST OF Code Value [p. 58]

3.3.4 Examples

The following example is completely made up. None of the mentioned code systems exist, and the scenario is admittedly rather strange. A code value for the hair color "ash-blond" in some local hair color code:

```
(CodeValue :value      "AB"
           :codeSystem "99hcc"
           :printName  "ash blond")
```

the translation into the official WHO approved *International Code for Hair Colors* (ICHC). ICHC does not have a code for "ash-blond" but it has "pale-blond." So we take that one.

```
(CodeValue :value      "10.2"
           :codeSystem "ICHC"
           :printName  "pale blond")
```

Now, what we have are two codes that both try to describe the same concept (i.e. what the physician has seen as the hair color). We have to build a concept descriptor that contains both code values, the original "ash-blond" and its translation "pale-blond" into ICHC.

```
(ConceptDescriptor
  :originalText "... the patient’s hair had an ashy-blondish color ..."
  :translations
     (SET
        (CodeTranslation :label "xlat-1-label"
           :term
              (Code-Value
                 :value      "AB"
```

```
                  :codeSystem "99hcc"
                  :printName  "ash blond"
               )
            :origin #null
         )
         (CodeTranslation
             :term
                (CodeValue
                   :value       "10.2"
                   :codeSystem "ICHC"
                   :printName  "pale blond"
                )
             :origin (ref "xlat-1-label")
         )
      )
  )
```

In this example the type definition is deliberatedly "violated" in that the code phrase was not used as the *term* component of the Code Translation. This demonstrates the type conversion [p. 18] feature of our type system. We can allow to send one related type for another.

Suppose, the CDC is conducting a study to correlate ear infection with hair color. The *Pilological Society of America* (PILS-A) just has agreed on an *Advanced Hair Color Code* (AVACC), which CDC is using for its study. This code is post-coordinated. It has the axes (1) base color (black, brown, blond) (2) gray-tone (none, slight, medium, strong) and (3) homogeneity (homogene, spotty, ... [here I could be more creative in my native language]). The translator guesses that "blond, slight, homogene" would fit best (although the original text didn’t say anything about homogeneity). So we add that other translation:

```
  (ConceptDescriptor
    :originalText "... the patient’s hair had an ashy-blondish color ..."
    :translations
       (SET
          (Code-Translation :label "xlat-1-label"
            :term
               (CodeValue
                  :value      "AB"
                  :codeSystem "99hcc"
                  :printName  "ash blond"
               )
            :origin #null
          )
          (CodeTranslation :label "xlat-2-label"
             :term
                (CodeValue
                   :value      "10.2"
                   :codeSystem "ICHC"
                   :printName  "pale blond"
                )
             :origin (ref "xlat-1-label")
          )
          (CodeTranslation
             :term
                (CodePhrase
                   (LIST :of "Code-Value"
                      (Code-Value
                         :value       "B001"
                         :codeSystem "PILS-AVACC"
                         :printName  "blond"
                      )
                      (CodeValue
                         :value       "G002"
                         :codeSystem "PILS-AVACC"
                         :printName  "slight gray"
                      )
                      (CodeValue
                         :value       "H001"
                         :codeSystem "PILS-AVACC"
                         :printName  "homogene"
                      )
                   )
                 )
             :origin (ref "xlat-2-label")
          )
       )
  )
```

Because the translation program *interXhair*^{TM} does not know about the local code "99hcc", it can only translate from the ICHC term.

The features *quality* and *producer* of a translation are not shown in the above example here.

The Concept Descriptor can also deal with coding exceptions. The distinction between "code without exceptions" and "code with exceptions" was proposed before and we should make sure that we capture the requirements that this proposal tries to address. An exception in this system of coding and translating occurs if some particular quality that was observed can not be coded in a particular coding system.

For example, 46 year old Jane Jammer comes into Dr Doolittle’s office with the complaint of an itchy sensation in her gut, but it is not quite painful. On the question where that sensation is located exactly, Mrs. Jammer points to her upper left abdomen but then draws a circle that covers about everything.

So Dr. Doolittle tries to code this chief complaint using a *Multiaxial Code for Primary Care Medicine* (PRIMAX). PRIMAX might have an axis for sensation (S) and location (L). The doctor is lucky to find 123 "ABDOMEN" as a fairly general descriptor for the location. But the doctor finds only "pain," "numbness," "tension," "heat," and "cold" as sensations. So where does the "itchy but not quite painful" sensation go into? Unfortunately this code does not come with the category *not otherwise classified* (NOC) *not otherwise specified* (NOS) or just *other* that many classification systems (like ICD) have. So, the physician can not code that chief complaint of his patient.

The physician writes down the following:

```
(ConceptDescriptor
  :originalText "... an ’itchy’ feeling in her ’guts’ that is not
                     quite painful ..."
  :translations
     (SET
        (CodeTranslation
           :term
              (CodePhrase
                 (LIST :of "CodeValue"
                    (CodeValue
                        :value       #other
                        :codeSystem  "PRIMAX"
                        :replacement "itchy feeling, not painful"
                     )
                     (CodeValue
                        :value      "L-123"
                        :codeSystem "PRIMAX"
                        :printName  "abdomen"
                     )
                  )
                )
           :origin #null
         )
     )
)
```

3.3.5 Outstanding Issues

The special value `#null` means a value `(NoInformation)` of the No Information [p. 27] data type without a null flavor. The special value `#other` stands for

```
(NoInformation :flavor "other")
```

In order to fully support this, we need canonical taxonomy of flavors of null.

In the above example, PRIMAX is a multiaxial code, it has sensation (S), location (L), and may be other axes, like timing (T), and the situation in which the problem occurs (W). PRIMAX (like SNOMED) does not require you to pick a value from every axis. So, no one knows what this #other in PRIMAX refers to, sensation? timing? work-relatedness?

It seems to be redundant to have a code phrase such as the following

```
(Code-Phrase
  (LIST
     (CodeValue
        :value      "S-001"
        :codeSystem "PRIMAX"
        :printName  "pain"
     )
     (CodeValue
        :value      "L-123"
        :codeSystem "PRIMAX"
        :printName  "abdomen"
     )
     (CodeValue
        :value      "T-032"
        :codeSystem "PRIMAX"
        :printName  "post prandial"
     )
     (CodeValue
        :value      "W-120"
        :codeSystem "PRIMAX"
        :printName  "pulling a carriage"
     )
  )
)
```

Because every code here is taken from the same code system PRIMAX, one would not need to specify PRIMAX as the code system for all those related Code Values.

It also seems as if a code phrase does only make sense in certain code systems. For example, in LOINC a code phrase is pretty useless if not contradictory to the (original) style of LOINC (that has been loosened up lately). In LOINC you would say

```
(CodeValue
   :value      "2703-7"
   :codeSystem "LOINC"
   :version    "1.0K"
   :print-name  "OXYGEN:PPRES:PT:BLDA:QN"
)
```

for the partial pressure of oxygen (pO_2)in an arterial blood sample. It is certainly wrong in LOINC to say the same in a phrase that first mentions pO_2 in NOS blood (`BLD`) and then adds to it the modifier that the specimen was really arterial blood.

```
(Code-Phrase
   (LIST :of "Code-Value"
      (Code-Value
         :value       "11556-8"
         :code-system "LOINC"
         :version     "1.0K"
         :print-name  "OXYGEN:PPRES:PT:BLD:QN"
      )
      (Code-Value
         :value       "BLDA"
         :code-system "LOINC-SYSTEM"
         :version     "1.0K"
         :print-name  "arterial blood"
      )
)
```

If the ability to form code phrases depends on the code system, the code system might define a syntax for literal expressions of those phrases, such as "M12345 F03847 D94578" which SNOMED apparently suggests.

On the other hand, some coding systems that do have modifiers (like HCFA procedure codes) do not necessarily specify a syntax to build code phrase literals.

Even codes that are not originally meant to be used in code phrases (like ICD9 used to be long time ago) did evolve to allowing formulation of code phrases. Today we see that certain ICD9 codes beg for a second code to specify the meaning more exactly.

We currently see such a drift towards multiaxiality within LOINC. LOINC originally distinguished between a glucose lab test and a glucose test-strip, and while LOINC defines

```
(Code-Value
  :value       "8736-1"
  :code-system "LOINC"
  :version     "1.0K"
  :print-name  "CARDIAC OUTPUT:VRAT:..:CARDIAC VENTRICLE.LEFT:FICK"
)
```

for the cardiac output measured using Fick’s principle (oxygen intake equals oxygen transport rate in blood). Recently, LOINC seems to allow the same thing to be said in another way:

```
(Code-Phrase
   (LIST :of "Code-Value"
      (Code-Value
         :value       "8741-1"
         :code-system "LOINC"
         :version     "1.0K"
         :print-name  "CARDIAC OUTPUT:VRAT:..:CARDIAC VENTRICLE.LEFT"
      )
      (Code-Value
         :value       "8825-2"
         :code-system "LOINC"
         :version     "1.0K"
         :print-name
            "HEMODYNAMIC MEASUREMENT METHOD:TYPE:...:CARDIAC VENTRICLE.LEFT"
      )
 =    (Code-Value
         :value       "FICK"
         :code-system "noLOINC"
         :version     "1.0K"
         :print-name  "Fick’s principle"
      )
   )
)
```

This is not quite right, because LOINC is still not multiaxial. You would have to guess that the third Code Value in the phrase is here to assign a value to the second Code Value, like "*method :*=`FICK`".

Sometimes we need to label specific parts in a code phrase. A code phrase is just a container of a flat sequence of code values. Language has deep structure (look at Chomsky's famous noun phrase (NP) and verbal phrase (VP))

Our data type is already quite complex. If we do a recursion of the EBNF form:

```
        CodePhrase ::= { CodeTerm };

        CodeTerm   ::= CodePhrase | CodeValue;
```

then we would be very powerful, but would also add a significant amount of complexity. We do not fear recursion here, but we do not want to create a super-powerful data type that will provide thousands of ways for people to abuse its power and hardly any idea about how to use the power properly.

Note that from the SNOMED camp there is probably support for an even more complex definition of the Code Phrase that would basically be a keyword-value structure containing small conceptual graphs. [cf. Spackman KA. Compositional concept representation using SNOMED: towards further convergence of clinical terminologies. *Proc Annu Symp Comput Appl Med Care.* 1998 Oct. p. 740-4.]

3.4 Technical Instances

There are two different modes of referring to technical instances. You can (1) *identify* an instance among other instances present in a set (e.g. identifying a record in a data base). For instances that are not immediately present, one can (2) *locate* that instance by *dereferencing* a pointer. However, there are many similarities between instances and pointers. It appears that those identifiers can have three levels of quality. They can be

1. unique (globally)
2. un-ravel-able
3. de-reference-able

Unique Identifiers

Suppose you are given two identifiers. What you can always do is to compare them literally (i.e. character by character.) Now, if it turns out that these identifiers are literally equal, what do you know? You know that they both refer to the same identical instance if and only if you can be sure that the literal match of both identifiers is not accidential because of some naming conflicts.

Through narrowing down namespaces we can achieve uniqueness of identifiers quite easily. This is for example why in computer programming local variables in procedures are safer than using global variables. The real important quality of uniqueness is that identifiers are *globally* unique. Global uniqueness is generally achieved by a structure defined in the following piece of BNF:

<identifier> ::= *<name>* *<namespace>*

<namespace> ::= *<identifier>*

Obviously this is a recursive structure, i.e. every namespace is itself identified by a name in its parent namespace. This recursion up the namespace hierarchy must somehow be terminated. This is done by assigning one globally unique namespace, where names are valid without the reference to another namespace.

The uniqueness of an identifier does not imply, however, that a given instance could not have several names. Thus, if you compare unique identifiers literally and you find that they do not match, you know nothing. Both identifiers can still refer to the same instance.

Un-ravelable Identifiers

An identifier is "unravelable" if we can analyze its pieces, and for each piece, we can find someone to talk to.

Internet domain names (DNS) are unravelable expressions. For example we can unravel the string "`falcon.iupui.edu`" from the right, where "`edu`" is maintained by Internic (the organization that assigns top level Internet domains). When the *Indiana University Purdue University Indianapolis* (IUPUI) registered its domain name "`iupui`" with the Internic, they had to name an official person who is responsible for "`iupui`". That person knows what "`falcon`" is.

ISO Object Identifiers (OID) are unravelable too. ISO OIDs are unraveled from the left. For example,

```
1.2.840.10008.421292.87828.333433.001
```

stands for

ISO (1) ISO member body (2) USA (840) DICOM Standard (10008) AGFA (421292) ...

The left most numbers are registered with gigantic organizations. Eventually, a company like AGFA gets a number allocated, say, 421292. It then creates machines where one of the machines has the number 87828. That machine allocates numbers to an imaging study (333433), that contains a series of images (001).

In unraveling an ISO OID we walk the path down basically the same way as with DNS names. DICOM has registered people with in the US member body of ISO (ANSI). AGFA has registered people to DICOM. They, or someone in the radiology department, could probably tell you that 87828 is the CT machine in the trauma center. Finally, the machine itself allocates identifiers at "computer speeds" to things like studies and images.

You can try out how it feels to unravel an OID using the information compiled by Harald T Alvestrand (http://www.alvestrand.no/objectid).

HL7 filler orders are somewhat unravelable. For example, you are given the filler order "`1234^OUTPATIENT.LAB`". If you could figure out what department the symbol "`OUTPATIENT.LAB`" referred to, then you could call them up, and ask them about item "`1234`".

As we can see, the quality that an identifier is unravelable is a result of the way the namespaces are managed. Both ISO OIDs and Internet domain names are organized through hierarchical namespaces.

De-referenceable Identifiers

An identifier is "dereferenceable" if there is a machinery that resolves those identifiers for you rather than requiring you to go the rather painful way of unraveling. For Internet domain names there is such a machinery dedicated to resolve names. I.e. the domain name service (DNS). The Internet name server next to you will resolve the address for you quite seamlessly. There is a whole infrastructure of domain name services, which is why it takes so long to get an answer from a DNS server if you typed in a wrong domain name: your DNS server asks another server that asks another server and so on.

For ISO OIDs there is no such easy way of dereferencing. In some cases there may be catalog services (e.g., X.500) that can resolve a subspace of the whole gigantic OID namespace.

A telephone number, however, is a perfectly unique and dereferenceable identifier if we start at the root of the namespace provided by the global telephone system. Fax numbers are usually written in a standardized way, where for instance "`+49308153355`" used to be my old fax and phone number in Germany, while "`+13176307960`" is my office phone number in U.S. All you need to do to dereference such a phone number is to pick up your phone, dial the prefix for international codes ("`+`"), dial the other digits and my telephone will ring.

Unified Resource Locators (URL) are another example of dereferenceable identifiers. For instance,

```
http://aurora.rg.iupui.edu/v3dt
```

is the version 3 data type project’s homepage. Your browser and the Internet does everything for you after you typed in this URL. URLs start with naming the protocol to use, the rest of the URL is a literal that the protocol is supposed to understand. For example, I can watch the same homepage as a local file using the URL

```
file:/home/schadow/public_html/v3dt/index.html
```

In general, for an identifier to be dereferenceable it need not be practically un-ravelable. For instance, a telephone number is for all everyday purpose not unravelable (only law enforcement is given this privilege). You may be able to figure out a country code (1 for U.S.) and an area code (317 for Indianapolis), but you will have a pretty hard time to find the number 6307960 in the phone book of Indianapolis.

The important point about dereferencing identifiers is that you do not get down to their "meaning" in the real 3D world through the process of dereferencing. I.e. unless you come into my office, you will never see my machine, "`aurora`", featuring the above homepage. And the machinery that dereferences URLs seamlessly does not bring you into my office. All you can do is to look at what the Internet/HTTP/Browser machinery brings to your screen as a result of dereferencing the URL identifier. Likewise with the telephone you can call me, but you cannot creep through the wire to see my telephone.

We therefore create two different data types for referring to technical instances, one for technical instance identifiers and another for technical instance locators.

3.4.1 Technical Instance Identifier

Technical Instance Identifier (TII)			
This data type is used to uniquely identify some entity that exists within some computer system. Examples are object identifier for RIM class instances, things like medical record number, placer and filler order number, service catalog item number, etc.			
component name	**type/domain**	**optionality**	**description**
root	ISO Object Identifier [p. 85]	**required**	This is the required field that guarantees the uniqueness of the identifier and that permits the origin of the identifier to be determined (un-raveled). This can be the only field in institutions that use OIDs for their internal object identifiers.
extension	Character String [p. 38]	**optional**	The extension can be used in case an institution uses non-OID identifiers locally and does not want to map every internal identifier to OIDs. Especially useful if the local identifiers are not purely numeric. **This field may never ever be send alone without the connecting root OID.**

HL7 identifiers for technical instances are to be unique. For identifiers to be unique we have to manage the global namespace. Most importantly every identifier must be explicitly linked to the root of the namespace hierarchy. Since HL7 has acquired a branch in the tree of ISO OIDs we are free to use OIDs in a similar way as DICOM uses OIDs heavily and directly.

In order to foster interoperability the technical instance identifier requires ISO Object Identifiers to be used. No other alternative unique identifier scheme is permitted. ISO Object Identifiers are very common, and sufficiently easy to acquire.

Many existing HL7 systems do not assign purely numerical identifiers for the technical instances in their realm. For instance they may use alphanumeric unique keys into any data file. We do not force people to adopt a pure OID scheme for identifiers.

HL7 can, however, assign OIDs to everyone who writes applications for HL7 and to everyone who maintains HL7 communications. On that basis, people are free to attach their own naming scheme to their standard OID. If they want, they may use OIDs in their realm, but they may also use free-form identifiers in the extension component.

Organizations can use OID that they already have acquired from elsewhere (e.g. through DICOM). HL7 assigned OIDs are not required. HL7 assigns OIDs as a service to its members and users, but does not require OIDs to root in the HL7 branch.

3.4.2 ISO Object Identifiers

ISO Object Identifier (OID)
The ISO Object Identifier is defined by ISO/IEC 8824:1990(E) clause 28.
PRIMITIVE TYPE

The ISO definition of Object Identifier reads as follows:

> 28.9 The semantics of an object identifier value are defined by reference to an **object identifier tree**. An object identifier tree is a tree whose root corresponds to [the ISO/IEC 8824 standard] and whose vertices [i.e. nodes] correspond to administrative authorities responsible for allocating arcs [i.e. branches] from that vertex. Each arc from that tree is labeled by an object identifier component, which is [an integer number]. Each information object to be identified is allocated precisely one vertex (normally a leaf) and no other information object (of the same or a different type) is allocated to that same vertex. Thus an information object is uniquely and unambiguously identified by the sequence of [integer numbers] (object identifier components) labeling the arcs in a path from the root to the vertex allocated to the information object.
>
> 28.10 An object identifier value is semantically an ordered list of object identifier component values. Starting with the root of the object identifier tree, each object identifier component value identifies an arc in the object identifier tree. The last object identifier component value identifies an arc leading to a vertex to which an information object has been assigned. It is this information object which is identified by the object identifier value. [...]

From ISO/IEC 8824:1990(E) clause 28

The following diagram shows part of the huge ISO Object Identifier tree referred to in the definition.

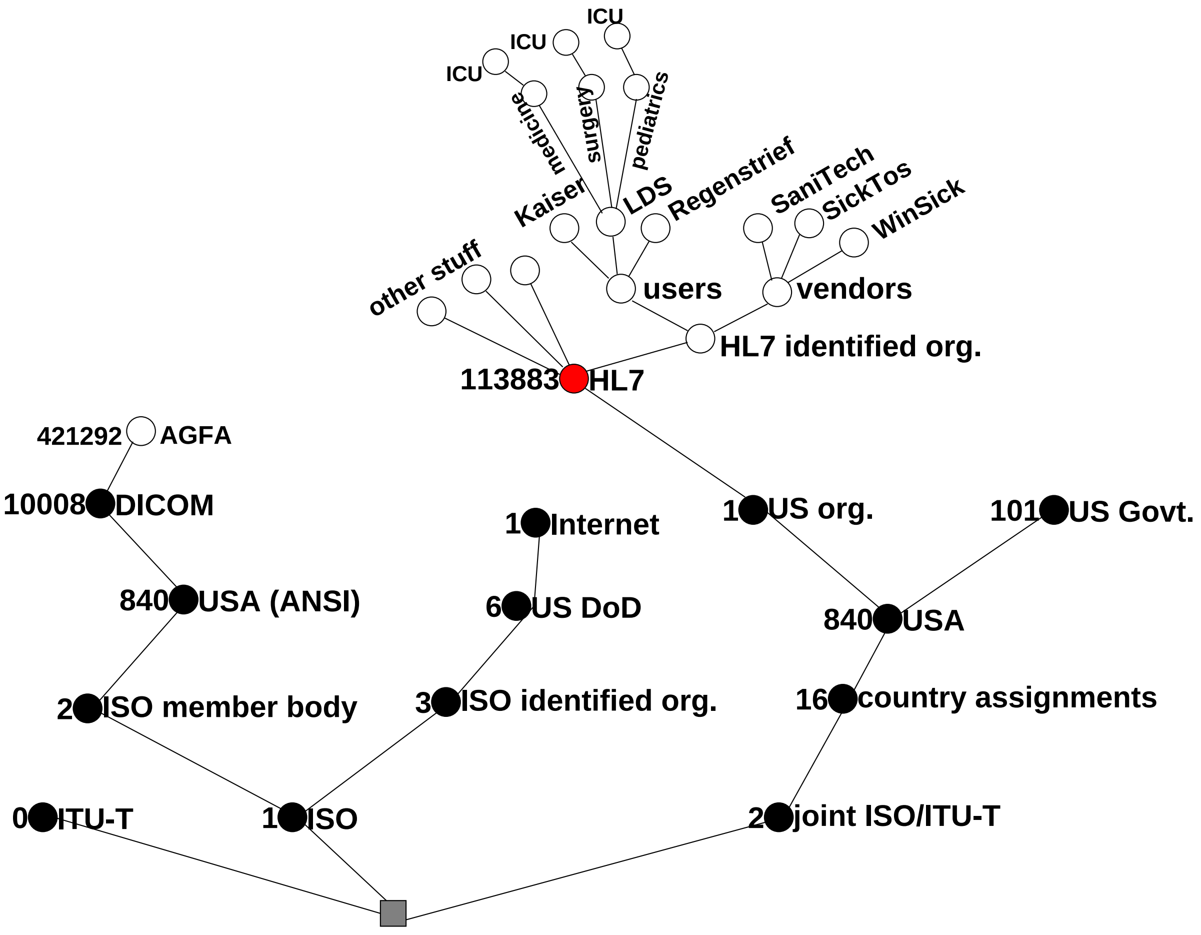


Figure 4: The the hierarchy of ISO Object Identifiers and how it could be used by HL7.

Rather than as a composite data type, we treat ISO Object Identifiers as primitives. However, through their semantic structure, there are a number of operations that can be performed with the object identifier, including test for equalness and subsumption (i.e. partial match from the left). Just like in DICOM, ISO Object Identifiers may be treated as character strings by the ITS layer.

How difficult will it be to *acquire* OIDs?

ISO Object Identifiers come with the blessing of being world-wide unique and endorsed by the *International Organization for Standardization* (ISO). At the downside, one might be afraid how difficult it will be for small vendors and users to make all the bureaucrats happy just in order to get one of such a unique Object Identifier.

The good news is that no HL7 vendor or user has to contact ISO in order to get an OID. OIDs are assigned hierarchically so that every OID can itself be reused as the basis for a large tree of other OIDs. As soon as you have one OID you are an assigning authority by yourself. No need for you to contact anyone else in order to issue other OIDs.

HL7 itself has acquired an OID recently. This makes HL7 an assigning authority. On the one hand, we may use OIDs for HL7 internal things. On the other hand we could have one branch for HL7 identified organizations. This branch could be subdivided into users and vendors.

A vendor who has acquired an OID could name all his HL7 related products machines, software, single installations of their software and so on as OIDs in their subtree.

For example, the Letter Day Saints (LDS) Hospital in Salt Lake City would have an OID at the user's side. They can, for example, subdivide their tree in pediatrics/medical/surgical departments where each of them may have an ICU subdepartment with its systems and subsystems and so on. The Idea is that everyone can do with its part of the subtree whatever they want. Regenstrief and Kaiser would have their OIDs to organize their namespace as they see fit.

The point is that you need to get only one OID from somewhere else. Once you have your first OID, you do with it whatever you want. It is just like you can design your directory hierarchy on your hard disk just as you want. You can stick to a convention, or you can do chaos, as you see fit.

How difficult will it be to *use* OIDs?

One may hesitate to use ISO Object Identifiers within a system because of the amount of memory they use up, in other words, the OIDs can become quite lengthy. Many legacy systems have their pain threshold as low as 8 characters for identifiers. An OID would not fit into 8 characters. For example, some instance in the LDS pediatric ICU might have the following OID:

```
  2.16.840.1.113883.4.1.123456.32.101.12345.54321
```

That is 44 characters. DICOM has set the maximal length to 80 characters. We will not specify any particular maximal length since length is a problematic concept for Object Identifiers and OIDs are meant to be unbounded.

But there is even a way to get around with only 8 characters. Here is how:

No one should have trouble *sending or receiving* those long OIDs. The problem with length is only about *storing* OIDs in data bases. Now, you can use an OID data base at your system that can handle long OIDs and that maps those to 8 byte base 64 strings. Those 8 byte strings allow you to enumerate a total of 64^8 = 281,474,976,710,656 different identifiers. This is 2.814 x 10^{15}, a thousand-trillion numbers. Suppose you would waste those identifiers at a rate of 1000 per second, your namespace would still not overflow in 8900 years!

What ISO Object Identifiers can and can not do

One might wonder whether it is possible to interpret OIDs in a globally agreed way. The Andover Working Group tried to design the OID namespace structure in a way that OIDs would not only *identify* instances but would also *classify* them.

So the question is: can we parse an OID and get any information from it? Can we learn anything about an instance just by looking at its OID? Things that we might what to find in an OID are: What Application? What Facility? What Department? What Country? What Location? Which Type? etc.

We have to review our goal: we wanted to design an unique identifier for technical instances. Uniqueness that comes through hierarchical structure of the namespace brings with it the quality of un-ravelability of identifiers. But the original meaning of "un-ravelable," was that unraveling an identifier is a painful and slow process. You use the phone, calling up ISO, ANSI, HL7, LDS, and so on until you have someone on the phone who is responsible for that number. Unraveling is nothing that a computer could do for you automatically. (Automatic unraveling would be dereferencing or resolving an identifier.)

Thus, in general, there is no way to impose any meaning on the parts of an OID.

However, owners of OIDs may "design" their namespace subtree in some meaningful way. For instance, Intermountain Healthcare could assign an OID to each of its institutions, the next level would contain departments. In each departments the number 1 would be the administrative section, number 2 would be the ICU, number 3 would be the lab, number 100 to 999 would be the normal inpatient wards, and so on.

Everyone is free to design and use his own OID structure to make decisions. However, no one outside would be forced to do the same structuring. Thus, Intermountain Healthcare could base it’s message routing heavily on the structures of their OIDs, but as soon as they receive something from the Utah State Dept. of Health or from the CDC, they would not necessarily be able to infer any meaning from the OIDs assigned by those other organizations.

Can the root part of the OID be implied by some context?

This really asks whether we can reduce the size of messages by setting any specific context, probably in the message header, which would be attached at the front of each incomplete OID that appears in the message.

Apart from reducing message length, this does not seem to be a particularly useful feature. ISO Object Identifiers do not support any left-side incompleteness. We probably need not bother.

The main benefits of the Technical Instance Identifier using ISO Object Identifiers are

- Simplicity (only two components!)
- Flexibility (OIDs are already quite flexible, the "extension" component gives you all the rest.)
- Interoperability (No worries for name clashes. No headache with local stuff. Actually, everything is local, but those localities are well organized in the overall OID system.)

3.4.3 Technical Instance Locator

Another kind of data type for technical instances is the Technical Instance Locator (TIL), which is a dereferencable identifiers, reference, or (technical) address. The Technical Instance Locator (TIL) is shaped similar to *Universal Resource Locator* (URL). That is TIL has the two components *protocol* and *address* where the format of *address* is determined by the protocol. Telephone number, e-mail address, and the locator for an image reference pointer would be of this data type.

Technical Instance Locator (TIL)			
This is a dereferencable locator for some instance. For example, a bunch of radiology images that can be retrieved on demand. A given instance of this data type may not be valid forever.			
component name	**type/domain**	**optionality**	**description**
protocol	Code Value [p. 58] for technical concepts	**required**	This mentions the protocol that can interpret the access string and can do something useful for the user to render the particular technical instance referred to. This may be spawning a WWW browser with a particular URL, fetching a DICOM image and show it, or opening a telephone connection to another party.
address	Character String [p. 38]	**required**	This is an arbitrary address string that must be meaningful to the protocol.

This data type is basically the URL. However. URLs are not maintained by HL7 and HL7 may need to have more freedom about defining its own protocols without adjustment to IETF needs. For example, we telephone numbers are semantically clearly Technical Instance Locators. A URL for telephone numbers does not exist, but it is conceivable how it would work. It would use an auto dialer to dial the telephone number put the called party on hold and signal to the human user that the line is opened. The human user would then pick up the phone and start talking. Likewise a URL for FAX data would initiate calls to send or retrieve telefax messages automatically.

Examples for values of the TIL type are:

```
(TIL
   :protocol (CodeValue
                 :value      "http"
                 :codeSystem "URL")
   :address  "//aurora.rg.iupui.edu/v3dt")

(TIL
   :protocol (CodeValue
                 :value      "ftp"
                 :codeSystem "URL")
   :address  "//radiology.rg.iupui.edu/outbox/1ad832nd84nf.jpg")


(TIL
   :protocol (CodeValue
                 :value      "mailto"
                 :codeSystem "URL")
   :address "your-boss@your-company.com")

(TIL
   :protocol (CodeValue
                 :value      "PHONE"
                 :codeSystem "HL7PROT")
   :address "+13176307960")

(TIL
   :protocol (CodeValue
                 :value      "FAX"
                 :codeSystem "HL7PROT")
   :address "+13176306962")
```

3.4.4 Outstanding Issues

We will still define as successor of the reference pointer (RP) to include the technical instance locator but also more information about the thing that is referred. This would also include an expiry date after which the locator can not be expected to be usable.

The use of the TIL for phone numbers needs more explanation and rationale.

The TIL may need to be wrapped in a History [p. 173] .

The TIL may need some "use code", to capture the qualifiers "business", "home", "cellphone", etc. for phone numbers. How does this "use code" generalize to other communication addresses? Why is it needed?

3.5 Real World Instances

We generally refer to things in the "real world" by giving them names. Assigning names to people, things and places are a public acts: the more people know some name, the more will later understand what is meant by that name. In archaic cultures, knowing the name of something meant having some power over it. Indeed, knowledge is power and without a name, we can not talk about things, we can barely think of things, and we can not collect knowledge about them. The record linking problem is a moderen example pointing out the importance of names. Names are the communicative handles over things.

Alternatively, instead of naming things, we can describe them. The problem with descriptions is that they refer to classes of everything that meets the description; but descriptions do not refer to individuals. Of course, descriptions can be so detailled that there happens to be no second alternative object in a given universe of discourse. Thus a description may identify an object.

As opposed to descriptions, a name is essentially an arbitrary token assigned to the object it refers to. Since assignment of names is an action, it is performed by some actor. In the real world many actors are entitled to assign names to entities. It thus happens that two or more things can be given the same name. Moreover, the association between a thing and its name is not substantial, thus, this association can be lost. Birth certificates, passports, or tags are artifacts aiming in substantiating the name-thing-association.

This specification covers the following kinds of names:

- Real World Instance Identifier [p. 93] (e.g., SSN, DLN, Inventory #, etc.)
- Postal and Residential Address [p. 101]
- Person Name [p. 111]
- Organization Name [p. 133]

Real World Instance Identifiers (RWII) are tokens designed to generate regular names, names that are handy and that have little ambiguity. Mostly those identifiers are designed to be easily computer-processable. The difference to a Technical Instance Identifier [p. 84] (TII) is that the TII naming scheme is tightly regulated, and that TIIs are supposed to never go through the hands of humans. Conversely, RWII does not regulate the naming scheme, and RWIIs are often tagged on things, issued on id cards, and are typed into information systems.

The Person Name specification must deal with all the richness, variability, and ambiguity, that the cultural elaborations of person names entail. Organization Names are very similar to person names, however, we simplify organization names drastically, since it was felt that organization names play a much less crucial role in health care than person names.

Addresses are also names for real world entities. The fact that locations tend to be extremely stable over a long period of time determines the structure of the address kind of names. Addresses determine locations by stepwise refinement of a scope (country - city - street - house - floor). Most scope-names have all the characteristics of names, i.e. arbitrarily assigned, non-descriptive, not unique. Apart from scope refinement all kinds of spacial descriptors can be part of an addres (e.g. right hand side, opposite side, north, east, etc.)

3.5.1 Real World Instance Identifier

External identifiers for real world people and things occur frequently. Examples for people identifiers are Social Security Number (SSN), Driver License Number, Passport Number, Individual Taxpayer Identification Number. Identifiers for organizations are, e.g., the federal identification number or the Employer Identification Number. The current approach in the RIM is to use the Stakeholder_identifier class for those numbers.

Here are some of those identifiers used in the U.S.

- Social Security Number (SSN and ITIN) - for U.S. persons;
- Employer Identification Number (EIN) - for U.S. corporations;
- ITIN (Individual Taxpayer Identification Number), like an SSN but issued by IRS for aliens not eligible for an SSN;
- Driver License Number (DLN) - for U.S. residents, are issued by the states, U.S. are used as identity cards.
- HIPAA Provider Identification Number - for U.S. healthcare provider
- HIPAA "Universal" (meaning "U.S.American") Health Identifier - if it will ever come.
- Inventory Numbers - for desks, computers, and coffee makers in everyone’s office
- Credit Card Numbers - for people and their CC accounts
- Medical Record Numbers - for a patient as the subject of a medical record
- Passport Number

Other countries may or may not have similar identifiers. The interesting point is that such identifiers are often used for other than the original purposes. For example, very few U.S. people care about whether you have a license to drive, but they do want your driver license number anyway in order to get hold of your identity (e.g., to trust your bank check.) The U.S. SSN may officially not be used by everyone, but that does not keep everyone from using it as a pretty reliable person identifier. Banks and employers must collect the SSN of their customers and employees (resp.) for tax purposes.

While many of such identifiers are assigned to people and organizations, what characterizes those numbers is not what they are assigned to, but who assigns them, how they are assigned, and how they are used. There is a need for such numbers to be assigned to real world instances other than people or organizations. Examples are things, such as devices and durable material (inventory numbers), lot numbers, etc.

The following challenges exist for exchanging real world instance identifiers:

1. "Communication Horizon" - if you communicate an identification number in-house, there is usually good understanding and no ambiguity. For inter-institutional communication there is possible ambiguity in the primary identifiers and the secondary identifiers for assigning authorities.
2. Information about assigning authorities is relevant or irrelevant depending on the scope of a message.
3. Systematizing identifier types and usage in an international context is difficult.

Organizations as assigning authorities

The following kinds of organizations assign real world instance identifiers:

- National governmental agencies (e.g., SSN, HCFA provider ID)
- State/Province governmental agencies (e.g., DLN)
- Professional organizations (e.g. AMA)
- Insurers, Banks, Credit Card Companies (e.g., Kaiser, BC/BS, VISA for their customers)
- Health provider organizations (e.g., Hospital Medical Record Numbers, Inventory Numbers.)
- Departments and other sub-organizations (e.g., special MRN rings for stat assignments.)
- non-formal units or task forces within an organization.(e.g. clinical trial enrollment number)

Considering health provider organizations (as the main users of HL7 messages,) we can distinguish three general cases where the assigning authority is treated slightly different:

1. National and state agencies' numbers are "well known," e.g. nobody ever wants to see the address and phone # of the U.S. Social Security Administration (SSN) or the Indiana Bureau for Motor Vehicles (DLN) in an HL7 message.

   Moreover, the identifier types themselves are an "institution" much more important than the assigning authorities. For example, the SSN data field will often times contain Individual Taxpayer Identification Numbers (ITIN) that are compatible to SSNs but are assigned by the IRS rather than the SSA. The distinction between SSN and ITIN is tricky and mostly irrelevant for HL7 users.

Professional organizations are usually treated as "well known." E.g., if you have a doctor’s medical license number valid for the U.S., you don’t need to communicate the details of the issuing organization (e.g. AMA.)

2. Insurers, Banks and Credit Card Companies are "third party" organizations that are external to health provider organizations. This means, most HL7 messages will want to add some minimal information about the assigning authority as an organization because those third party organizations are neither "well known" nor do they belong to any one provider organization.

3. Provider organizations and their sub-units. These are the issuers of the vast majority of numbers communicated in everyday messaging. For all "in-house" messages, the assigning authority is the same or closely related to the HL7 user. So, there is no need to communicate much information about that organization.

   For external communication, however, the assigning organization needs to be identified with more detail. Generally, the less routinely messages are sent to a particular external recipient the more detail information about assigning authorities is appreciated.

4. Finally there are cases where the same organization assigns different numbers of the same type. For example, patient identifiers are issued for routine care, but the same health care organization runs several clinical trials where patients get separate identifiers or enrollment numbers. Thus, the same organization that runs different trials will want to build partitions of the overall set of assigned identifiers (sub-namespaces.)

Identifier types and their use

We intuitively know that there are different types of identifiers and that we want to keep track of the identifier type. The first identifier type that comes to mind in a U.S. context is the Social Security Number (SSN). This example shows two difficulties that any "typology" of identifiers runs into and must deal with:

1. Semantics (meaning) and pragmatics (use) of one type of identifier may be completely different and not even related. For example, the meaning of the SSN is that it identifies every U.S. person’s social security record. But the SSN is only in 5% (estimated) of all uses cases related to a person’s social security matters. Much more often (40%), the SSN is used as a person’s taxpayer’s identification number (by the IRS or by withholding agents, such as employers, banks, or mutual fund/IRA services.) Most health provider organizations use the SSN as a pretty good national person identifier (40%). In addition all kinds of companies collect SSNs from their customers for various purposes.

2. Identifier type concepts do not easily translate between different realms (e.g. countries.) Take Social Security Numbers (SSN) for example: most countries that have a nationally organized social security system will have social security numbers. However, as noted

> above, the purpose of collecting SSNs in the U.S. health care industry is not social security, but person identification. Germany has SSNs too, but nobody uses the German SSN as a general person identifier. German SSNs are exclusively used in communications with the German social security administration about genuine social security issues.

The same case can be made for the Driver License Number. In Europe, driver licenses are primarily used as a certification to run a motor vehicle, and thus in 90% of the cases shown to police officers and highway patrols. In the U.S. the situation is completely different: here, more than 50% of driver license checks occur in bars and night clubs to gain entrance and to be served alcoholic beverages. Another 20% of driver license are shown when people write checks. Another 20% fall on miscellaneous identity checks, while in less then 10% of the cases a traffic policeman will be the one to see your driver license. Clearly, in the U.S. driver licenses are identity cards. In Europe, people have government issued identity cards. However, the numbers are much less often recorded.

In conclusion, designing a terminology of "identifier types" is difficult and has to account for the difference between what an identification number *is* and what it is used for.

Naively one would like to post-coordinate identifier type and country/state code, however, as noted above an (SSN, US) is something completely different than an (SSN, DE), which means that identifier type and country are not really orthogonal. The better approach therefore seems to be to assign separate identifier types for each type and country of identifier, that is, to pre-coordinate the identifier type code. Thus the U.S. SSN would be uniquely identified and no other country’s SSN would be assigned to the same type. An example of a completely pre-coordinated identifier type code is shown in the following table.

Examples of a pre-coordinated terminology of identifier types

code	type	country	state	issuer	notes
001	SSN	US			national person identifier
002	DLN	US	AB		Alabama
003	DLN	US	AL		Alaska
004	DLN	US	AZ		Arizona
...	...	...	...		...
053	DLN	US	WN		Wisconsin
054	med. license	US		AMA	License for U.S. certified Internists.
008	med. license	DE.	BW	LGM	Baden-Württemberg
009	med. license	DE	BA	LGM	Bayern
010	med. license	DE	B	LGM	Berlin
...	...	...	...	...	...
024	med. license	DE	SWH	LGM	Schleswig-Holstein
011	citizen id	DE			the number on the ID card (German "Personalausweis.")
012	citizen id	DK			
013	citizen id	FR			
...	...	...	...	...	...
123	patient-id	any	any	any	medical record number, requires issuing auth.
124	inventory	any	any	any	inventory number, requires issuing auth.

However, there is a downside to pre-coordinated non-hierarchical codes with meaningless identifiers. While these codes comply to the currently touted "good vocabulary practices," the administrative systems that will be using those codes will not be able to make much use from those identifier types. The problem is most obvious when it comes to U.S. driver licenses or German medical licenses. These are issued on a state-level (sub-national governmental agencies.) Therefore, there are 50 codes for U.S. driver licenses and 16 codes for German medical licenses. While this detail is rarely needed, the simple test for "is this a driver license?" is much more difficult than with a simple code "DLN" with the state post-coordinated.

Those will be the issues that need to be considered when defining the terminology for identifier types. While they are not a core part of this harmonization proposal, they do affect the current information model design and this extended documentation is necessary for the record.

Definition in the Information Model

The definition of the Real World Instance Identifier (RWII) is based on a class by the same name in the HL7 Reference Information Model (RIM.) This is so because there is an association between the RWII and an organization as an "assigning athority" of the identifier. This presents a methodological challenge: the RWII should be available as a data type but the data type is associated with an information model class.

The Unified Modeling Language correctly makes no difference between an attribute’s data type and a class, any class can be used as a data type for an attribute. The HL& Modeling and Methodology Committee has decided to accept the notion of a "DMET", that is a Common Message Element Type (CMET) useable in the RIM as a data type. That way we avoid a large bundle of associations connecting from every other class to the RWII class. The following figure shows the new structure of the RIM as of June 1999.

Figure 5: The Real_world_instance_identifier as an information model class. "Users" of this class may not associate to it but will refer to the RWII DMET as a data type, as shown in the Stakeholder class' "real_id" attribute.

Definition of the DMET

The DMET definition of the RWII data type is as follows

Real World Instance Identifier (RWII) *DMET*			
An identifier for a "real world instance". A real world instance is any person, organization, provider, patient, device, animal, or any other thing that some organization recognizes and assigns an identifier to. Examples are Social Security Number, Driver License Number, Inventory Number, HCFA Provider ID, Medical Record Number. Typically, real world instance identifiers are assigned and reused outside of HL7 communication. These identifiers tend to be less reliable than Technical Instance Identifiers that are assigned and maintained exclusively by HL7 communication systems. Other classes use this class not by associations but by declaring attributes of type "RWII."			
component name	**type/domain**	**optionality**	**description**
value_txt	Character String [p. 38]	mandatory	The character string value of the identifier. For example the character string "123-45-6789" for a U.S. Social Security Number."
type_cd	Code Value [p. 58]	mandatory	A code representing the type of identifier. For example, codes to represent the US National Provider ID, US National Payor ID, US Health Care ID, medical record number, social security number.
qualifier_txt	Character String [p. 38]	conditional	Information used to limit the applicability of a real world instance identifier, such as the state or province in which the identifier is valid. Use and interpretation depends on the type_cd.
valid_tmr	Interval [p. 166] of Point in Time [p. 148]	optional	The time range in which the identifier is valid. May be undefined on either side (effective or expiration).
assigned_by	Organization (RIM class CMET)	conditional	The assigning authority of the identifier if not implicit in the type_cd. The Organization CMET used here is likely to be very terse.

While the value_txt is always a mandatory part of a real world identifier, the qualifier_txt must, may, or must not be valued depending on the identifier type_cd. This is independent of whether a precoordinated or a postcoordinated identifier type coding scheme is used. As the above table suggests, there is no way to completely precoordinate identifier type codes when the issuer organizations are not "well known" (e.g., providers, insurers.)

For example, the state of the U.S. driver license is either precoordinated in the identifier type_cd or it is post-coordinated in the qualifier_txt. The qualifier_txt can be used for patient identifiers to allow issuing authorities to maintain multiple namespaces (e.g., for multiple clinical trials.)

The actual use of the real world instance identifier should not be coded in the type_cd but should be given implicitly through establishing many more attributes in a many classes that have the data type RWII (a DMET.) For example, rather than pushing all stakeholder identifiers up to the highest level, the Stakeholder class should have an identifier only for such identifiers as SSN, EIN, ITIN, passport number, person id. Medical record numbers (patient id) should be declared as an attribute of the Patient class. Provider license numbers should be declared in the Individual_health_care_provider class, etc.

The identifier issuing authority is a conditional component of the real world instance identifier. The organization will not be mentioned in a message for "well known" issuers (e.g., SSN, DLN, etc.) The organization will be mentioned by a brief object stub for in-house communication. For third-party organizations and for inter-enterprise communication, there will be more information given for the issuing organization.

Finally, it must be noted that technical instance identifiers (TII) are a much more economic structure to identify patients and things in HL7 messages for routine use. After external identifiers (RWIIs) have been exchanged once, follow-up messages should generally suffice with TIIs.

Medical Record Numbers (MRN) as used in the world of Paper Medical Records are another example for such real world instance identifiers. Note that in the computer world, we would not need MRNs, since we could use Technical Instance Identifiers [p. 84] (TII) to refer to computerized medical records. However, Wes Rishel and I think that as a rule of thumb, TIIs should not be communicated through human middlemen in order to keep reliability in their correctness high. Thus, as long as MRNs are typed in by clerks and other people, one should separate them from TIIs.

3.5.2 Postal and Residential Address

The old HL7 address data types (AD, XAD) regarded an address as a data structure where each component had a special role. For instance, AD distinguished ZIP, city, state, country, street, and other parts of the address.

Over time people discovered more information elements that could be known about an address and added those elements as components to the address data type. Those additional components where county, census tract, etc. Those information items would normally not appear on mailing labels and one would not necessarily ask for them if oue would go visit someone under a given address.

On the other hand it turned out that there are a number of information elements that do appear on mailing labels which are nevertheless rare and therefore remained unclassified. For instance, U.S. military addresses may have a unit designation "UNIT 2050" instead of a street and instead or in addition to a city. The name of a ship (e.g. "U.S.S. Enterprise") can appear instead of a city.

Internationally there are other address parts that may exist in one country but may be unknown in another country. For example, in U.S. addresses one finds directional codes like "N", "S", "W", and "E", which are essential to find a given address in one city. Those direction codes are unknown, for instance, in Germany.

Robin Zimmerman and Joann Larson have compiled an analysis of U.S. and some international addresses based on information of the universal postal union (http://www.upu.int/) (UPU). This work reinforces the observation that there are so many different kinds of address parts that creating a fixed data structure where every part has its slot is impractical. See also examples of world wide addresses (http://www.upu.int/addressing/AN/AN.pdf) as published by the UPU. There is also an australian standard that defines the pieces an address is made up of.

Another problem with the old address data types was that they ordered the parts of an address by the meaning of that part. The most important use case for address information, however, is printing a mailing label. In order to generate a mailing label it doesn't matter what the emaning of the different parts of an address is, as long as those parts appear at the appropriate place on the label.

The placement of address parts, however, depends on the country. For example, while in U.S. and most European addresses the ZIP code appears somewhere at the end, Japanese ZIP codes are written at the very top. In fact, Japanese addreesses are writen in the reverse direction: from the most general locator tho the specific locations, with the name of the recipient appearing at the end.

Even in addresses of the north western part of the world there are such differences as to how ZIP code and city are placed. In Germany and most European countries, for instance, the ZIP code is placed in front of the city, while in England, the ZIP code appears after the city name on a separate line. In the U.S. the zip code follows the city and usually the state code. In most European countries, special country codes (different from ISO 3166 country codes) are written before the ZIP code (separated from the ZIP code by a dash). In U.S. and England country codes appear at the end. In Great Britain, however, the ZIP appears even after the country designator, whereas in the U.S.A. the country code appears at the very end.

In short, layout and meaning of address parts are independent (orthogonal) issues, but the address data type must take care of both. The focus, however, is not on the meaning of the parts, but on the layout. Although we could define a semantically very fine-grained address part classification, those would be impractiacl to use with a large majority of existing information systems that do not make those fine grained semantic distinctions. There are simply too many different address parts and too many different country-specific variations, that may or may not really correspond.

Thus, focusing primarily on the layout of address labels is a way to establish a greatest common denominator for interoperability. System A might store addresses in 5 lines. System B might distinguish ZIP code, country, state and a street line. System C might distinguish a house-number on the street line (common in Germany or Holland). System B can use system C’s addresses and A can use addresses from both B and C.

It is still a problem how system C can find a house number in the street-line or how system B can identify a street-line in a list of lines received from system A. Rather than forcing everyone to make the most fine-grained distinction we require those systems who make the distinctions to deal with the less distinctive addresses.

Postal and Residential Address (AD)			
This Address data type is used to communicate postal addresses and residential addresses. The main use of such data is to allow printing mail labels (postal address), or to allow a person to physically visit that address (residential address). The difference between postal and residential address is whether or not there is just a post box. The residential address is not supposed to contain other information that might be useful for finding geographic locations or doing epidemiological studies. These addresses are thus not very well suited for describing the locations of mobile visits or the "residency" of homeless people.			
component name	**type/domain**	**optionality**	**description**
purpose	Code Value [p. 58]	**optional**	A purpose code indicates what a given address is to be used for. Examples are: prefered residency (used primarily for visiting), temporary (visit or mailing, but see History [p. 173]), preferred mailing address (used specifically for mailing), and some more specific ones, such as "birth address" (to track addresses of small children). An address without specific purpose code might be a default address useful for any purpose, but an address with a specific purpose code would be prefered for that respective purpose.
bad address flag	Boolean [p. 24]	**optional**	Indicates that an address is not working. Absence of a status means "unknown" status, i.e., that is’t presumably a good address. If the flag is set explicitly to *false,* it means that this address has been proven to work at least once.
value	LIST OF Address Part [p. 104]	**mandatory**	This contains the actual address data as a list of address parts that may or may not have semantic tags.

Address Part (ADXP)			
This type is not used outside of the Address [p. 103] data type. Addresses are regarded as a token list. Tokens usually are character strings but may have a tag that signifies the role of the token. Typical parts that exist in about every address are ZIP code, city, country but other roles may be defined regionally, nationally, or on an enterprize level (e.g. in military addresses). Addresses are usually broken up into lines which is indicated by special line break tokens.			
component name	**type/domain**	**optionality**	**description**
value	Character String [p. 38]	**mandatory** exception: for line break tokens.	The value of an address part is what is printed on a label.
role	Code Value [p. 58]	**optional**	The role of an address part (if any) indicate whether an address part is the ZIP code, city, country, post box, etc.

Purpose Codes for Address

Short	Long	Meaning
R	RES	**r**esidency used primarily to visit an address.
P	PO	**p**ostal address used to send mail.
T	TMP	**t**emporary address visit or mailing, but see History [p. 173]
B	BRTH	**b**irth address CDC uses those for child immunization.
...		

Role Codes for Address Parts

Short	Long	Meaning
L	LIT	**l**iteral **this is the default role code**
K	DEL	delimiter stuff, printed without framing whitespace. Line break if no value component provided.
C	CNT	**c**ountry
T	CTY	city (**t**own)
E	STA	state ("E" as in French **é**tat, which should reconcile the French who have to use "E" for their "departements")
Z	ZIP	**Z**IP code
H	HNR	**h**ouse number (aka. "primary street number", however, it is not the number of the street, but the number of the house or lot alongside the street.)
A	ADL	**a**dditional locator can be a unit designator, such as appartment number, suite number, but also floor. There may be several unit designators in an address to cover things like: "3rd floor, Appt. 342". This can also be a designator that points away from the location, rather than specifying a smaller location within some larger one. Example is Dutch "t.o." to mean "opposite to" for house boats.
S	STR	**s**treet name or number
ST	STT	street type (e.g. street, avenue, road, lane, ...) (probably not useful enough)
D	DIR	**d**irection (e.g., N, S, W, E)
P	POB	**P**.O. Box
...		

Examples

Please note that the person name is not part of our address type even though it is mentioned by UPU and Joann/Robin’s list.

A U.S. address

1028 Pinewood Court
Indianapolis, IN 46240
U.S.A.

```
(Address (LIST
  (AddressPart :value "1028 Pinewood Court")      ; LIT is the default role
  (AddressPart :role "DEL")                       ; DEL’s value is newline by default
  (AddressPart :value "Indianapolis" :role "CTY")
  (AddressPart :value ", " :role "DEL")           ; DEL comes w/o extra space
  (AddressPart :value "IN" :role "STA")
  (AddressPart :value "46240" :role "ZIP")
  (AddressPart :role "DEL")                       ; DEL’s value is newline by default
  (AddressPart :value "U.S.A." :role "CNT")))
```

A German address

Windsteiner Weg 54A
D-14165 Berlin

```
(Address (LIST
  (AddressPart :value "Windsteiner Weg 54A") ; LIT is the default role
  (AddressPart :role "DEL")                  ; DEL’s value is newline by default
  (AddressPart :value "D" :role "CNT")
  (AddressPart :value "-" :role "DEL")       ; no whitespace before and after
  (AddressPart :value "14165" :role "ZIP")
  (AddressPart :value "Berlin" :role "CTY")))
```

White Space Rules

Address labels contain white space. The white space rules used in typestetting are not trivial. In general two words are separated by white space. An interpuction mark, like a komma or period follows directly to the preceding non-whitespace stuff, but those marks are always followed by whitespace. Dashes are not surrounded by whitespace at all. Note the whitespace rules do not really exist for languages such as Thai or Japanese where white space is basically not used. However, you can always simply ignore whitespace, which is why Thai and Japanese are easier to print. In any case, neither Thai nor Japanese would have whitespace where it was not allowed in Latin script.

The difficult whitespace rules can, for the purpose of the Address data type be broken down into only six precise rules:

1. White space never accumulates, i.e. two subsequent spaces are the same as one. Subsequent line breaks can be reduced to one. White space around a line break is not significant.

2. Literals may contain explicit white space, subject to the same white space reduction rules. There is no notion of a literal line break within the text of a single address part.

3. Leading and trailing explicit whitespace is insignificant in all address parts, except for delimiter (`DEL`) address parts.

4. By default an address part is surrounded by implicit white space.

5. Delimiter (`DEL`) address parts are not surrounded by any implicit white space.

6. Leading and trailing explicit whitespace is significant in in delimiter (`DEL`) address parts.

This means that all address parts are generally surrounded by white space, but white space does never accumulate. Delimiters are never surrounded by implicit white space and every whitespace contributed by preceeding or succeeding address parts is discarded, whether it was implicit or explicit. For example, all of the following variants

```
(lit "1028") (lit "Pinewood Court")
(lit "1028 ") (lit "Pinewood Court")
(lit "1028") (lit " Pinewood Court")
(lit "1028 ") (lit " Pinewood Court")
(lit "1028    ") (lit "     Pinewood Court")
```

are printed the same way:

```
"1028 Pinewood Court"
```

with only one white space between "1028" and "Pinewood Court".

A DEL address part is a delimiter, and would never be framed by implicit white space. As noted above, a comma is always followed by white space, but this whitespace would have to be part of the value part of the delimiter. HL7 systems do not have to enforce all those typographical rules. For example, all of the following variants

```
(lit "Indianapolis") (del ", ") (lit "IN")
(lit "Indianapolis ") (del ", ") (lit "IN")
(lit "Indianapolis") (del ", ") (lit " IN")
(lit "Indianapolis ") (del ", ") (lit " IN")
```

are printed the same way:

```
"Indianapolis, IN"
```

with no white space before the comma and only one white space after the comma, i.e. the white space that has been provided *lit*erally in the delimiter value string. This *lit*eral space could have been missing, as in the following cases

(*lit* `"Indianapolis"`) (*del* `","`) (*lit* `"IN"`)
(*lit* `"Indianapolis "`) (*del* `","`) (*lit* `"IN"`)
(*lit* `"Indianapolis"`) (*del* `","`) (*lit* `" IN"`)
(*lit* `"Indianapolis "`) (*del* `","`) (*lit* `" IN"`)
(*lit* `"Indianapolis"`) (*del* `","`) (*lit* `"     IN"`)

which are printed all the same way:

```
"Indianapolis,IN"
```

without the space after the comma. This is not good typographic style, but it is not enforced by HL7 rules. No space is wanted around dashes, such as in European addresses:

(*cnt* `"D"`) (*del* `"-"`) (*zip* `"12200"`) (*cty* `"Berlin"`)
(*cnt* `"D "`) (*del* `"-"`) (*zip* `"12200"`) (*cty* `"Berlin"`)
(*cnt* `"D "`) (*del* `"-"`) (*zip* `"12200"`) (*cty* `" Berlin"`)

which are printed all the same way:

```
"D-12200 Berlin"
```

The DEL address part does not need any value for a DEL's value is a line break by default. Note that our whitespace rules apply nicely to line breaks, since a line break makes trailing white space of the previous line redundant and leading white space of the subsequent line is correctly removed too.

Further Examples

The following is another U.S. address with maximal tagging of the address parts:

1001 W 10th Street RG5
Indianapolis, IN 46202
U.S.A.

```
(Address (LIST
  (AddressPart :value "1001" :role "HNR")
  (AddressPart :value "W" :role "DIR")
  (AddressPart :value "10th" :role "STR")
  (AddressPart :value "Street" :role "STT")
  (AddressPart :value "RG5" :role "LIT")
  (AddressPart :role "DEL")
  (AddressPart :value "Indianapolis" :role "CTY")
  (AddressPart :value ", " :role "DEL")
  (AddressPart :value "IN" :role "STA")
  (AddressPart :value "46202" :role "ZIP")
  (AddressPart :role "DEL")
  (AddressPart :value "U.S.A." :role "CNT")))
```

The instance notation shows how different the new address type is compared with the old HL7 AD/XAD types.

This address type is an interesting construct: It is kind of the inverse of a record data structure. In a record, we have a bunch of slots that may or may not contain data. In this data type we have a bunch of data that may or may not be assigned slots.

XML ITS

It is especially interesting to see how this data type maps into XML. An automatic mapping (as the one used for the HIMSS demo) would create a very long unreadable XML. But the reason for the popularity of XML is that markup can be added gently to a basically "human readable" text. XML wise a much nicer represenation would be:

```
<Stakeholder.addr TY="AD">
  1001 W 10th Street RG5
  Indianapolis, IN 46240
  U.S.A.
</Stakeholder.addr>
```

the contents of this address could now be refined:

```
<Stakeholder.addr TY="AD">
  1001 W 10th Street RG5
  <A P=CTY>Indianapolis</A>, <A P=STA>IN</A> <A P=ZIP>46240</A>
  <A P=CNT>U.S.A.</A>
</Stakeholder.addr>
```

note that in the above represenation we at least allowed address part roles to occur as XML attributes. If DTDs were not used, one could even create a nicer representation if we turn the role codes into XML tags.

```
<Stakeholder.addr TY="AD">
  1001 W 10th Street RG5
  <CTY>Indianapolis</CTY>, <STA>IN</STA> <ZIP>46240</ZIP>
  <CNT>U.S.A.</CNT>
</Stakeholder.addr>
```

Actually the address data type is an example for the paradigmatic use case of XML: a bunch of data that may or may not be further marked up. It would be very odd if we would not use XML in this classic way for this classic use case.

Outstanding Issues

Should we allow for address part values other than mere Character Strings? Especially, should we allow for code values? Using code values seems to make sense for things like country code and state. Using a code table for state or countries is of course safer and allows to process addresses into groups.

While this is possible in general, we have three problems:

1. The data type definition and all of the instances would become more complex, since we have to define the AddressPart.value as a type choice between CharacterString [p. 38] and CodeValue [p. 58] (or even ConceptDescriptor [p. 72] !)

2. While there are codes for U.S. states and countries (e.g., ISO 3166 Country Code (http://www.unece.org/trade/rec/rec03en.htm)) those codes are not used uniformly. There are two forms to abbreviate U.S. states, e.g., the Commonwealth of Massachusetts can be "`MA`" or "`Mass.`". While the ISO country code is suggested for international use, there is a

long tradition in Europe to abbreviate countries in a different code (same that is used for country stickers on cars.) Thus, the ISO code for Germany is "`DE`" but "`D`" is used all over Europe.

Since there are different code tables in use one might even require the Concept Descriptor [p. 72] data type to account for the translations. This is a considerable overhead, for what use?

3. The use case of codes in addresses is very limited. If a receiver really wants to rely on those codes, we set up a number of requirements that did not exist before. (1) the address part must be tagged with an explicit role, (2) the right code must be used by the sender. The use case to code addresses is very localized, which means, the coding of address parts may be needed in one application but it is not needed in many others. In order to print labels and visit people, coded address parts are not essential.

We probably do not whant to make the address data type any more complex than it already is. HL7 should certainly not *impose* more requirements to code certain address parts. It just seems not to be a widely demanded use case, an *a priory* argument for coded address parts, which could offset the lack of use cases, seems to not exist.

However, there is one powerful way in which the simpler address data type defined here can meet the needs of those who would like to have coded address fields: type casting.

Through type casting a message would be valid even though the sender put a CodeValue [p. 58] , or ConceptDescriptor [p. 72] in place of a CharacterString. This means, a sender, who does code address parts, is able to send his coded address parts to a peer, who also prefers to receive coded address parts where possible. Thus, an implementation may behave as if the address data type would be defined in a more complex way.

The point is, we don’t have to make the HL7 specification more difficult to understand and implement for those who do not want this extra feature of coded address parts and still allow those who want to deal with the extra work to go ahead and do it. This is another example where implicit type casting in a well defined type system proves extremely useful: the canonical specification can remain simple, and still extra requirements can be supported in a compatible way!

3.5.3 Person Name

The HL7 v2 person name data types (PN, XPN) have basically the same problems as the data type for addresses [p. ??] . I.e., they try to make slots for data so that whatever name parts exist must be fitted in one of the available slots. This has the same disadvantages: that name part types do not classify in a simple and interchangeable way throughout all cultures, but still everyone must use the same classification. Second problem is that the meaning of a name part and the positioning of a name part are orthogonal (independent) aspects of a name. As an additional problem, person names may occur in different ordering and some name parts are or are not used depending on the use case (e.g., formal vs. familiar style).

The decisions made here were informed by the following references:

1. Bidgood DW Jr, Tracy WR. *In search of the name.* Proc Annu Symp Comput Appl Med Care, 1993; p. 54-58.

2. Bidgood DW Jr, Tracy WR. *ANSI HISPP MSDS: COMMON DATA TYPES for harmonization of communication standards in medical informatics.* Final Draft. 10/30/1993. Available as Postscript (http://www.mcis.duke.edu/standards/HISPP/MSDS/CommonDataType1102.ps) or Word (http://www.mcis.duke.edu/standards/HISPP/MSDS/CommonDataType1102.doc).

3. Hopkins R. *Strategic short study: names and numbers as identifiers.* CEN TC251. Available as PDF (http://www.centc251.org/SSS/NandN/SSSNandN18.pdf) or Word (http://www.centc251.org/SSS/NandN/SSSNandN18.rtf). Note especially Appendix B: *National Name Forms* by Arthur Waugh, Australia.

4. Anonymus. *A Study on names in the US and in the Netherlands* Available here (http://www.mcis.duke.edu/standards/HL7/localization/HL7NetherlandsNames97-198.htm).

5. This conference call was based on a worksheet that summmarizes some earlier discussions.

We first present the proposed data structure for person name and then we will show examples, discuss ramifications, and justify why this particular design has been chosen.

Data Type Specification for Person Name

The Person_name is a RIM class as of June 1999. This class is correctly associated with the class Person and the multiplicities of this association allow one person to have multiple names. A second association ("is_used_by") to the class Statkeholder allows a person name to be scoped to some organization (or even another individual person.)

Within this RIM class Person_name, there is an attribute that indicates what purpose a given name is to be used for ("reason_cd") Most people in the world will have one name that is currently used. The following table is the Control Query recommendation to PAFM for a mandatory vocabulary for Person_name.reason_cd. We also suggest to rename this attribute to "purpose_cd".

Name Purpose Codes

SYMBOL	SHORT	DESCRIPTION
normal	`N`	The name normally used. May be restricted through validity time intervals.
license	`L`	Name not normally used, but registered on some record, license or other certificate of professional or academic credential, but that is not normally used (includes birth certificates, school records, degrees & titles, and licenses.)
artist	`A`	An artist’s pseudonym includes "stage name", writer’s name.
indigenous	`I`	Indigenous or tribal names, such as existing abong native Americans and Australians.
religious	`R`	Name adopted through practice of religion. For example, "Father Irenaeus," "Brother John," or "Sister Clementine" are religious names that persons adopted through entering an order or assuming a religious office or both.

Note that name purpose codes apply to an entire name that usually consists of several of the name parts described below.

There is also a way to specify the validity time of a name.

This class also contains an attribute "nm" which contains a single name variant as a list of person name parts that may or may not have semantic tags. This person name data type (PN) is defined as follows:

Person Name (PN)
This type used in the RIM class Person_name that will be developed from the class Person_alternate_name of RIM 0.88 jointly with PAFM. Person names consist of tagged Person Name Parts [p. 113] . Typical name parts that exist in about every name are given names, and familiy names, other part types may be defined culturally.
LIST OF PersonNamePart [p. 113]

Person Name Part (PNXP)			
This type used in the Person Name data type only. Each person name part may have a tag that signifies the role of the name part. Typical name parts that exist in about every person name are given names, and familiy names, other part types may be defined culturally.			
component name	**type/domain**	**optionality**	**description**
value	Character String [p. 38]	**mandatory**	The value of a name part.
classifiers	SET OF Code Value [p. 58]	**optional**	Classifications of a name part. One name part can fall into multiple categories, such as given name vs. familiy name and name of public records vs. nickname.

Note that the Person Name (PN) data type is different from the Person_name class. The data type is not a CMET or DMET of the class but is used by the class as the data type of one of its attributes. The naming overlap is to indicate that this HL7 version 3 PN data type is the successor of the HL7 version 2 PN data type, while the Person_name class can be understood as the successor of the version 2 XPN data type.

Name Part Classifiers

SYMBOL	**SHORT**	**DESCRIPTION**
Axis 1 This is the main classifier. Only one value is allowed.		
given	`G`	Given name (don't call it "first name" since this given names do not always come first)
family	`F`	Family name, this is the name that links to the genealogy. In some cultures (e.g. Eritrea) the family name of a son is the first name of his father.
prefix	`P`	A prefix has a strong association to the immediately following name part. A prefix has no implicit trailing white space (it has implicit leading white space though). Note that prefixes can be inverted.
suffix	`S`	A suffix has a strong association to the immediately preceeding name part. A prefix has no implicit leading white space (it has implicit trailing white space though). Suffices can not be inverted.
delimiter	`D`	A delimiter has no meaning other than being literally printed in this name representation. A delimiter has no implicit leading and trailing white space.

Axis 2 Name change classifiers decribe how a name part came about. More than one value allowed.		
birth	`B`	A name that a person had shortly after being born. Usually for familiy names but may be used to mark given names at birth that may have changed later.
unmarried	`U`	A name that a person (either sex) had immediately before her/his first marriage. Usually called "maiden name", this concept of maiden name is only for compatibility with cultures that keep up this traditional concept. In most cases maiden name is equal to birth name. If there are adoption or deed polls before first marriage the maiden name should specify the last family name a person acquired before giving it up again through marriage.
chosen	`H`	A name that a person assumed because of free choice. Most systems may not track this, but some might. Subsumed in the concept of "chosen" are pseudonyme (alias), and *deed poll*. The difference in civil dignity of the name part is given through the `R` classifier below. I.e. a deed poll creates a chosen name of record, whereas a pseudonym creates a name not noted in civil records.
adoption	`C`	A name that a person took on because of being adopted. Adoptions may happen for adults too and may happen after marriage. The effect on the "maiden" name is not fully defined and may, as always, simple depend on the discretion of the person or a data entry clerk.
spouse	`M`	The name assumed from the partner in a marital relationship (hence the "`M`"). Usually the spouse’s familiy name. Note that no inference about gender can be made from the existence of spouse names.
Axis 3 Additional classifiers. More than one value allowed.		
nick	`N`	Indicates that the name part is a nickname. Not explicitly used for prefixes and suffixes, since those inherit this flag from their associated significant name parts. Note that most nicknames are given names although it is not required.
callme	`C`	A callme name is (usually a given name) that is preferred when a person is directly addressed.
record	`R`	This flag indicates that the name part is known in some official record. Usually the antonyme of nickname. Note that the name purpose code "license" applies to all name parts or a name, whereas this code applies only to name name part.

initial	`I`	Indicates that a name part is just an initial. Initials do not imply a trailing period since this would not work with non-Latin scripts. Initials may consist of more than one letter, e.g., "Ph." could stand for "Philippe" or "Th." for "Thomas".
invisible	`0` (zero)	Indicates that a name part is not normally shown. For instance, traditional maiden names are not normally shown. "Middle names" may be invisible too.
middle	`MIN`	Emphasizes that a name part is "the middle name" in the classic U.S. American First-Middle-Last name scheme. This classifier may only appear once in the entire name and may only be ascribed to the second *given* name part. No other use is permitted. Note that this tag is optional and completely redundant since the second of two given names can always be assumed to be "the middle name". It has been adopted only to satisfy public demand.
weak	`W`	Used only for prefixes and suffixes (affixes). A weak affix has a weaker association to its main name part than a genuine (strong) affix. Weak prefixes are not normally inverted. When a weak affix and a strong affix occur together, the strong affix is closer to the its associated main name part than the weak affix.
Axis 4 Additional lassifiers for affixes. Usually only one value allowed per affix. Classification does not try to be complete.		
voorvoegsel	`VV`	A dutch "voorvoegsel" is something like "van" or "de" that might have indicated noblety in the past but no longer so. Similar prefixes exist in other languages such es Spanish, French or Portugese.
academic	`AT`	Indicate that a prefix like "Dr." or a suffix like "MD" or "PhD" is an academic title.
professional	`PT`	Primarily in the British Imperial culture people tend to have an abbreviation of their professional organization as part of their credential suffices.
noblety	`NT`	In Europe there are still people with noblety titles. German "von" is generally a noblety title, not a mere voorveugsel. Others are "Earl of" or "His Majesty King of ..." etc. Rarely used nowadays, but some systems do keep track of this.

White Space Rules

Names contain white space. The white space rules used in typestetting are not trivial. In general two name parts are separated by white space. An interpuction mark, like a komma or period follows directly to the preceding non-whitespace stuff, but those marks are always followed by whitespace. Dashes are not surrounded by whitespace at all. Note the whitespace rules do not really exist for languages such as Thai or Japanese where white space is basically not used. However, you can always simply ignore whitespace, which is why Thai and Japanese are easier to print. In any case, neither Thai nor Japanese would have whitespace where it was not allowed in Latin script.

The difficult whitespace rules can, for the purpose of the person name data type, be broken down into the following precise rules:

1. White space never accumulates, i.e. two subsequent spaces are the same as one.
2. Literals may contain explicit white space subject to the same white space reduction rules.
3. Except for *prefix*, *suffix* and *delimiter* name parts, every name part is surrounded by implicit white space. Leading and trailing explicit whitespace is insignificant in all those name parts.
4. Delimiter name parts are not surrounded by any implicit white space. Leading and trailing explicit whitespace is significant in in delimiter name parts.
5. Prefix name parts only have implicit leading white space but no implicit trailing white space. Trailing explicit whitespace is significant in prefix name parts.
6. Suffix name parts only have implicit trailing white space but no implicit leading white space. Leading explicit whitespace is significant in suffix name parts.

This means that all address parts are generally surrounded by white space, but white space does never accumulate. Delimiters are never surrounded by implicit white space, prefixes are not followed by implicit white space and suffixes are not preceeded by implicit white space. Every whitespace contributed by preceeding or succeeding name parts around those special name parts is discarded, whether it was implicit or explicit.

Examples

Irma Jongeneel, of HL7 the Netherlands, has many nice ramifications in her name, so we will dwell a little bit on her name. Irma has two given names "Irma" and "Corine". In her childhood her family name was "de Haas". Then Irma married Gerard Jongeneel. In Holland both spouses can choose to use either or both of their familiy names in arbitrary order. For the public records Irma chose the combination "Irma Corine Jongeneel-de Haas". But we know her by the name "Irma Jongeneel", i.e. for casual cases she assumed the family name of her spouse. But if Irma would have to show up in a court of law and her name was cited, she would be called "Irma Corine de Haas e.g. Jongeneel" where "e.g." stands for "echtgenote van" meaning "spouse of".

Let’s write down the variants that we know now in the familiar instance notation [p. 22] .

First the name by which we know her

Irma Jongeneel

```
(PN
  (PersonNamePart :value "Irma"
    :classifiers (SET given record))
  (PersonNamePart :value "Jongeneel"
    :classifiers (SET family record spouse)))
```

Just as with the address we have to take care about spacing. When the name is to be printed we usually have the name parts separated by white space. But there are notable exceptions which we will encounter in the following example.

The following is the name of her marriage record (?)

Irma Corine Jongeneel-de Haas

```
(PN
  (PersonNamePart :value "Irma"
    :classifiers (SET given record))
  (PersonNamePart :value "Corine"
    :classifiers (SET given record))
  (PersonNamePart :value "Jongeneel"
    :classifiers (SET family record spouse))
  (PersonNamePart :value "-"
    :classifiers (SET delimiter))
  (PersonNamePart :value "de Haas"
    :classifiers (SET family record birth)))
```

Note that the dash "-" is printed without leading and trainling white space. This is signified by the flag *delimiter* in the name classifier set. We know this flag already from the from the Address data type. Since names never have line breaks, this line break feature does not exist with delimiters in person names.

Voorvoegsel

There is a problem with the "de" that is classified as a *voorvoegsel* in dutch. Another very common *voorvoegsel* is "van" as in "van Soest". This Dutch "van" is not actually a noblety prefix, although it sounds like it used to be one. Such prefixes exist in many languages, including, French, German, and Portugese.

The problem with such prefixes is that they belong to exactly one other name part, e.g., "Haas". In Dutch the part "Haas" of "de Haas" is called the *significant part* of that family name, since it is significant for alphabetic sorting. Since "de" can not occur without "Haas" and "Haas" will not occur without "de" both are linked stronger than "de Haas" and "Jongeneel".

One way to handle this associativity is through nesting. With parentheses we could write "(Irma (de Haas) Jongeneel)" to show that "de" and "Haas" are associated stronger than the other parts. However, nesting is costly as it leads to significant additional complexity in the data type definition. Not that nesting is a bad idea *per se*. However, since the nesting depth appears to be limited to three levels, the generality of nesting seems to not outweigh the simplicity of a simple linear list.

There are other ramifications though, such as prefixes that consist of more than one part such as in French "Eduard de l'Aigle". Here "de l'" is one prefix that consists of two parts and that connects to the significant part without spacing. To make things more complex we have to realize that "de l'Aigle" is in fact a contraction of "de-la-Aigle". But we decide not to deal with this kind of lexical variations. It is probably safe to consider "de l'" as *one* prefix that binds strongly to the following significant name part.

Thus we could go without nesting by using special name part flags "*prefix*". Prefix means that this name part binds strongly to the following name part and we consider it to bind without space. Let's try how that feels:

de Haas

```
(PN
  (PersonNamePart :value "de "
    :classifiers (SET prefix))
  (PersonNamePart :value "Haas"
    :classifiers (SET family)))
```

Note that "de " contains a literal space. Alternatively we could define flags for prefix-with-space and prefix-no-space, but this would just make things more complex. As a rule we say that name part prefixes associate without space to the following name. If a space is required, it must be included in the name part. See the white space rules above [p. 116] .

Eduard de l’Aigle has a prefix that includes no space

Eduard de l’Aigle

```
(PN
  (PersonNamePart :value "Eduard"
    :classifiers (SET given))
  (PersonNamePart :value "de l’"
    :classifiers (SET prefix))
  (PersonNamePart :value "Aigle"
    :classifiers (SET family record)))
```

Inversion

This method is challenged when we want to capture a inverted name form such as "Haas, de, Irma" used in a phone book or in bibliographies. Here we lose the strong association between to the prefix."de" and the its significant name "Haas". The prefix is postponed after the significant name "Haas", there is even an intermittent comma, and, to make things even worse, the spacing of "de" is different ("de" vs. "de "). It’s a matter of finding the most elegant solution. You can always argue about elegance of course.

Haas, de, Irma

```
(PN
  (PersonNamePart :value "Haas"
    :classifiers (SET family))
  (PersonNamePart :value ", "
    :classifiers (SET delimiter))
  (PersonNamePart :value "de "
    :classifiers (SET prefix inverted))
  (PersonNamePart :value ", "
    :classifiers (SET delimiter))
  (PersonNamePart :value "Irma"
    :classifiers (SET given)))
```

Here we say that the prefix "de " (with trailing space!) is *inverted*. The computer knows now that the prefix is associated with some preceeding stuff. The rule is: **An inverted prefix associates to the nearest preceeding name part that is not a delimiter.** Furthermore, the rule for printing the name is: **Trailing literal white space is to be removed from inverted prefixes**.

For Eduard de l’Aigle this works likewise:

Aigle, de l’, Eduard

```
(PN
  (PersonNamePart :value "Aigle"
    :classifiers (SET family))
  (PersonNamePart :value ", "
    :classifiers (SET delimiter))
  (PersonNamePart :value "de l’"
    :classifiers (SET prefix inverted))
  (PersonNamePart :value ", "
    :classifiers (SET delimiter))
  (PersonNamePart :value "Eduard"
    :classifiers (SET given)))
```

To completely cover all ramifications we can further undo the contraction "de l’A..." to "de la":

Aigle, de la, Eduard

```
(PN
  (PersonNamePart :value "Aigle"
    :classifiers (SET family))
  (PersonNamePart :value ", "
    :classifiers (SET delimiter))
  (PersonNamePart :value "de la"
    :classifiers (SET prefix inverted))
  (PersonNamePart :value ", "
    :classifiers (SET delimiter))
  (PersonNamePart :value "Eduard"
    :classifiers (SET given)))
```

However, this decomposition and contraction of "de la <vowel>" to "de l’<vowel>" and vice versa is outside the scope of HL7. This is rarely taken proper care of even in phone books or bibliographic databases so that hardly any HL7 application will need to care.

Echtgenote van, née, geb.

As we said earlier, when Irma shows up in a court of law, she might be called

Irma Corine de Haas e.g. Jongeneel

```
(PN
  (PersonNamePart :value "Irma"
    :classifiers (SET given record))
  (PersonNamePart :value "Corine"
    :classifiers (SET given record))
  (PersonNamePart :value "de "
    :classifiers (SET prefix)))
  (PersonNamePart :value "Haas"
    :classifiers (SET family record birth)))
  (PersonNamePart :value "e.g."
    :classifiers (SET prefix weak))
  (PersonNamePart :value "Jongeneel"
    :classifiers (SET family record spouse))
```

The "e.g." behaves pretty much like a prefix. It is not "significant" it associates with the following name part. The difference is that the association is *weak*. A weak association of a prefix or suffix means that the prefix might be dropped. It is still a prefix, which means that it moves wherever the following name part moves, but a weak prefix could be omitted.

Note that a weak prefix may be followed by a (strong) prefix, such as in "Gerard Jongeneel e.g. de Haas". Note also that if a weak prefix is followed by a name part which in turn is followed by an inverted (strong) prefix, the inversion would be undone by insertion of the (strong) prefix between the weak prefix and the significant name part. Contemplate "Jongeneel, Gerard e.g. Haas, de" as an example.

In "Claudine de l'Aigle née Dubois" and "Dorothea Schadow geb. Riemer" "née" and "geb." formally behave just like the "echtgenote van", i.e. they are weak prefices. However, note that the semantics is reversed. *Echntgenote van* means "spouse of" while *née* and *geborene* means "born" in French and German respectively.

Claudine de l'Aigle née Dubois

```
  (PN
    (PersonNamePart :value "Claudine"
      :classifiers (SET given record))
    (PersonNamePart :value "de l'"
      :classifiers (SET prefix)))
    (PersonNamePart :value "Aigle"
      :classifiers (SET family record spouse)))
    (PersonNamePart :value "née"
      :classifiers (SET prefix weak))
    (PersonNamePart :value "Dubois"
      :classifiers (SET family record birth))
```

The semantic difference between *née* and *e.g.* is not important since the classification of name parts into *birth* vs. *spouse* are non-ambiguous.

Nicknames

Let’s play a little bit with nicknames. I know Bob Dolin as "Bob", but at HL7 he is enrolled as "Robert Dolin" and on papers he calls himself "Robert H. Dolin". This is no big deal, since we have three distinct name forms that we decided to threat as separate Person names without trying to relate those name parts accross the variants.

Bob Dolin, **Robert Dolin**, or **Robert H. Dolin**

```
(PN
      (PersonNamePart :value "Bob"
         :classifiers (SET given nick))
      (PersonNamePart :value "Dolin"
         :classifiers (SET family))))

(PN
      (PersonNamePart :value "Robert"
         :classifiers (SET given))
      (PersonNamePart :value "Dolin"
         :classifiers (SET family))))

(PN
      (PersonNamePart :value "Robert"
         :classifiers (SET given))
      (PersonNamePart :value "H."
         :classifiers (SET given initial))
      (PersonNamePart :value "Dolin"
         :classifiers (SET family)))))
```

we did not classify the person name variants here, since this would open up another can of worms. It almost seems like there is a gradual scale of formality which tells which of the various person names to use.

Degrees of formality may be relevant, but are not yet handled in the HL7 data type. Other examples are: sloppy (Kiki), familiar (Kathy), nick (Kathrin), of record (Katharina) highly official (Ekatharina). We need input from Japan on that. Note also the "Bob Dolin" example above.

Let’s take Woody Beeler. Woody is known as "George (Woody) W. Beeler" in the HL7 membership data base. This parenthesis is an interesting construct that we might want to cover a bit more semantic and a bit less literal. The way Woody would pronounce this example is probably: "My name is George W. Beeler, but call me Woody." The parentheses are just a style to print the name badge. Actually the HL7 name badge looks like:

Woody

George W. Beeler

We do not allow line breaks in person names, instead of literal parenthesis or line breaks, we suggest a semantic markup using the *callme* name part classifier.

George (Woody) W. Beeler

```
(PN
   (PersonNamePart :value "George"
      :classifiers (SET given))
   (PersonNamePart :value "Woody"
      :classifiers (SET callme))
   (PersonNamePart :value "W."
      :classifiers (SET given initial))
   (PersonNamePart :value "Beeler"
      :classifiers (SET family)))
```

Two different applications could now use the same name variant to produce a name badge for an HL7 meeting and to print the HL7 membership directory. The rule for the badge application is: if there are "*callme*" name parts, print those in big and fat, and print all the other names below, except those names that are classified only as "*callme*". For the electronic membership directory the rule would be: print all names in order and use put *callme*-only name parts in parentheses.

Incomplete Classification

Let’s take some example where we just can’t classify the names. Consider "Iketani Sahoko". Of course, if you know some Japanese you will know that Sahoko is a Japanese female and "Iketani" is her familiy name. But let’s assume you don’t know that `:-)`. All you have is an unconscious girl wo has the name "Iketani Sahoko" printed (in latin letters) somewhere on her purse.

Iketani Sahoko

```
(PN
   (PersonNamePart :value "Iketani")
   (PersonNamePart :value "Sahoko"))
```

You now send this name without any classifier. The point is that you can not tell which one is the given name and which one is the familiy name. If you guess from the order (given name = first name) you are wrong. So, if in doubt, why being forced to guess? Of course, most data bases will force you to guess. But this wild guess can be done by the receiving HL7 interface just as well as by a unknowledgeable human. Later, when you learn more about your ptient, you can enter the correct classification:

Iketani Sahoko

```
(PN
   (PersonNamePart :value "Iketani"
     :classifiers (SET family))
   (PersonNamePart :value "Sahoko"
     :classifiers (SET given)))
```

HL7 v2.3 Compatibility

The XPN data type of HL7 version 2.3.x may serve as a validation to see what other name types or name part types may be needed. Of course, there is also the issue of compatibility between version 2 and version 3 of HL7.

One problem with mapping those name type codes between v2.3 and v3.0 is that our new person name type is structurally different from the old one. It is not possible, therefore, to simply reuse those codes without further thoughts.

The following table shows v2.3.x person name type codes. The right most column determines whether a code stand for an inherent meaning of a name (part) or for its purpose.

HL7 v2.3 XPN name types.

code	meaning	comments
A	alias	purpose, a person uses different aliases or pseudonymes in different contexts (i.e. when refering to himself as an author of a book, an actor, your friend, a customer in a bank, or a patient in a hospital.
L	legal	purpose, this is the name of public record (if any) Such records do not exist in all countries. In Germany legal names definitely exist, I am not so sure about the U.S.
D	display	purpose: for the purpose of "displaying"; however, this is quite vague. See below.
M	maiden name	inherent meaning, but there are also quite pragmatic implications. See below.
C	adopted	inherent meaning
B	name at birth	inherent meaning
P	name of spouse (name taken from)	inherent meaning

U	unsepcified	?? (obsolete)

The first issue is that the old person name had a bunch of fixed slots and a name type code affecting the interpretation of data found in all slots. Our new type has name parts wich are individually classified and it has a purpose code for name variants which affect all name parts of the name variant. The semantics of the name parts, i.e. what those parts *are*, is described entirely in the name part classifiers. Each name variant has a certain use case, purpose or context.

We have not retained the term "alias," for three reasons. First, one main assumption of our new approach to person names is to support different name variants, where every variant is baiscally an alias for a person. Thus there is no need to further qualify that. Second, the term "alias" has a negative connotation (e.g., only thieves and other bad guys need aliases.) Third and finally there are different kinds of pseudonymes that we may want to indicate positively, i.e. artist's names (writer and stage names), indigeous (tribal) names, and religious names.

In opposition to aliases, in some countries there are legal acts of name changes. In Australia, for instance, this is called "deed poll".

In Germany such name changes happen under exceptional conditions only and are always subject to official recording. The naming system in Germany is quite tightly regulated and you are not supposed to use any other name, except in certain situations where one would expect pseudonymes (e.g., book authors, actors, etc.)

In the U.S., however, name changes seem to be more frequent than in Germany and the naming system is less regulated as in Germany. One issue that one would need to clarify is the meaning of "legal" name. Legal name, obviously, has different meanings in different countries, depending on how the naming system is regulated.

The concept of display name was vague all along. The question is what display? The whole idea of names is that they are "displayed" on paper, computer screens, and in spoken language. The use case of display names thus is not clear. Basically there is no longer a need to have a name type "display name" in our new person name type. This is so, because we no longer distort the natural (or purposeful) ordering of the name parts by requiring name parts to be put in different slots. Name parts occur in some order that is defined or selected by someone, either the holder of that name or the computer system, or the citation style guide, etc.

Some names are used in Licenses or other accreditations and it is quite important to record the name as such. Examples are: school records, graduation certificates, license to practice a profession, etc. Notably, women who had a Doctoral degree were the first ones who assumed double names in Germany many decades ago. The reason was that their dissertations and certifications were issued for their maiden names. Later on, when those women married they would have lost their certifications by switching their family names entirely.

In many cases, keeping a name history is enough. However, the license name type allows one to indicate the reason why a certain name is still kept in the history, i.e., in this case, because it is mentioned in a license or record.

Maiden name, name at birth, name of spouse, adopted name, and the like.

This was a very difficult discussion, where a lot of arguments were exchanged but where people also said they could not even see the issue being so lively discussed.

Let's put this into historical perspective.

In versions 2.1 and 2.2 of HL7 there was no name type code at all, and the only place a "maiden" name was even mentioned was "PID-*mother's maiden name*". There was obviously no place to specify the patient's maiden name. This seemed to be somehow less of a problem in the U.S., but it was definitely a problem in Germany, which is why HL7 Germany redefined mother's maiden name to *patient's* maiden name.

Then came the name type code, and with it came the maiden name type code. The meaning of which was clear at that time, since there was just the maiden name and adopted name. It probably was not quite clear what would happen with a female that was adopted at 5 years, had a family name before and switched the family name through adoption and later married and switched the name again. We had a way to express the name she had after adoption, we were able to specify the name befor marriage, which in this case are the same! Two ways to specify the same name, but on the other hand, there was no way to specify neither the name *before* adoption, nor the name *after* marriage. Which is pretty odd, but, again, didn't seem to matter very much.

The famous Dutch name change initiative that started with a Sermon by John Baptist in summer 1997's meeting in San Francisco (or was it Tampa?), was the major driving force for bringing in "birth" name and "spouse" name types. As far as I know, the rationale was *not* to address the oddities mentioned in the last paragraph. Rather, the issue was that "maiden" seemed to imply "female before marriage" or even stronger cultural connotations. Since the people of the Netherlands have long had a very reasonable and free culture, the Dutch did away with those sexist traditions long before the rest of the world even realized the issue.

So the driving force behind "birth" name was to open up the narrow sense of "maiden". In that sense, "birth" was clearly meant to *subsume* "maiden".

The "spouse" name type on the other hand was meant as kind of the antonyme of "birth". The above examples around Irma Jongeneel are an extensive description of the dutch naming system which essentially explain why "birth" and "spouse" name types are so important in the Netherlands. It is all because a married (or otherwise officially associated) couple of persons (not necessarily of opposite gender), will sort of combine their family names while both names remain as independently useful family names. That's why birth name would get the "birth" classifier and the name of the spouse would get the "spouse" classifier.

From that perspective it seemed like "maiden" was subsumed by "birth", as a way to express the same concept with less sexist connotations.

But this was everything else than agreed to by everyone.

It turned out that the dutch reform has created more different notions than was originally expected. For example, again, what happens if someone changes his/her name before marriage? We finally decided that "maiden" and "birth" should not be merged, in parts, because "maiden name" is a cultural entity that may not exist in the Netherlads but still exist in many computer systems.

We made the observation that the above mentioned name types have different "directions" of meaning in time. They do not so much express what any name part is semantically, since family names are family names, but they try to capture how names come about. Dawid added, that those name types not only capture how names came about, but also, how names ceased to be used.

In the "ancient" U.S. name system of the 1950s and the German name system that losened up only recently the issues were simple. For instance, my wife’s name is "Dorothea Schadow" but her maiden name is "Riemer".

```
        Riemer <---MAIDEN|
  -----------------------+------------------------------> lifetime
                         |CURRENT---> Schadow
```

If we mention the maiden name of my wife, we indicate that this maiden name, "Riemer", was used for her before she assumed my family name, "Schadow", through marriage. So her current name is "Schadow" and will remain "Schadow" for the unforseeable future. Her family name *was* "Riemer" but no longer so. Now, it is just her maiden name. Thus, "maiden" name seems not to explain how the name "Riemer" came about, but it tells how the name part "Riemer" ceased to be used.

From the perspective of this very traditional naming scheme "maiden" and "current" is all you need to distinguish. And indeed most existing information systems are build based on this traditional misconception. No matter how strongly we may insist in this through our data base design, this is not how the world really works.

Since "maiden" is a term routed in the traditional patriarchal system, we can define "maiden" name as:

A "maiden name" is the surname of a woman before she marries.

at lest, this is what Webster’s has to say about "maiden name". Clearly, this notion appears archaic today. But still ADT system’s data bases, data entry forms and even application logic sometimes is built on this misconception.

Again, the Dutch people are the avant-garde of a more reasonable approach to looking at things. In the dutch naming system the "directions" are different, as Irma’s example showed that "maiden" is not an issue here:

```
|BIRTH---> de Haas
-----------------------+-------------------------------> lifetime
                       |SPOUSE---> Jongeneel
```

In the Dutch system, all name parts point forward. The name types explain how name parts came about, not how they ceased to be used.

From that perspective, "maiden" and "birth" do have different meanings. In the Dutch system the entire concept of "maiden name" simply does no longer exist. In Germany and the U.S. it still exists.

One could assume that maiden marks a name that ceased to be used, but this position seems to be no consensus. At the most I would open up the concept of "maiden name" to be less sexist so that I would like to see the definition to read as follows:

> A maiden name is a name part that a person had immediately before this person’s first marriage and that was given up due to that marriage.

By "marriage" I understand any kind of "culturally accepted personal association between human beings." This is open enough to include the wildest things as long as they are accepted in that culture (not necessarily accepted in other cultures). This includes homosexual marriages, religous (non-civil) marriages civil (non-religious) mariages; simply anything that causes someone to give up some of his/her name parts.

This is not just semantic talk. Practical connotations to a name part classified as "maiden" would be "don’t use it", except in special circumstances or with special prefixes.

What happens if someone get’s married and does not change her/his name?

From my perspective this is simple: "maiden name" simply does not apply.

However one can argue the other way: since "maiden" means young unmarried girl, you do have a maiden name even though you might have never gave up your name. Notably every maiden would have just a maiden name. Every unmarried person would have only a maiden name. Here it all depends on whether we think of names as slotted parts or as tagged parts. If name parts are slotted in data fields, the maiden name of a maiden is duplicated:

Pippi Langstrumpf

```
(SlottedName
  :given-name   "Pippi"
  :current-name "Langstrumpf"
  :maiden-name  "Langstrumpf")
```

In our new system, however, we tag names without duplications:

Pippi Langstrumpf

```
(PN
  (PersonNamePart :value "Pippi"
    :classifiers (SET given))
  (PersonNamePart :value "Langstrumpf"
    :classifiers (SET family maiden (current))))
```

What it all boils down to is the following problems:

- How do we map to and from slotted name structures?
- Do we have to adjust our model 100% to those flawed name categories that do not even hold in those cultures where they are most used? If so, how?

We gradually assumed the following rationale: birth name is the name you have at birth. Maiden name is the name you have just before your first marriage. An "Adoption name" is a name you have since you have been adopted (Beware of the ambivalence with "adopt**ed** name").

The immediate question becomes: what happens when you marry a second time? What if you are adopted after you first married (this can be done in some countries)? For me the question is, how many reasons of name changes do we have to capture? When is it enough to just keep a history of names?

- How many different events in a life do we want to recognize as having special name codes?

The answer is proably: "it depends". In Some cultures becoming a widow is a reason for a name change. In others you might change names as you give birth to children. You might also change names as you enter a religious community (e.g., as you become a monk, or a pope :-) Do we want to keep track of all this? Probably, it all depends.

For HL7 we have to stick to practical use cases. However, if we design the name data type according to a majority of existing information systems, we would still get stuck with the "first-m.i.-last" name pattern. A lot of the argument about maiden name was due to existing systems that either require a certain input or give a certain output. What should we do?

In general, we can recommend to consider only using the Dutch system, where we have a

1. name part at *birth*.
2. name part assumed through *adoption* (name of adopting parent)
3. name part assumed through *deed poll* (free change of name)
4. name part assumed through marriage (name of *spouse*)

Except from birth name, all other name change events may happen in arbitrary order and may repeat. All the rest is covered in a history. When you have a new name and you want to map to an old-stlye slotted name do the following to determine the maiden name:

1. If there has been no change of family name since birth, use that one and only family name at birth as the last name.
2. If a name part in question is taken from a *spouse* do **not** use this as a maiden name.

In other words, the maiden name is the family name in the history that was not assumed from spouse. Dealing with adoptions and deed polls is difficult, however, those things are not taken care of by the usual slotted name types anyway, so why bother?

The only strong rationale to keep maiden name is because mapping from a traditional slotted name structure to the new name style is difficult. With a "maiden name" you don't actually know whether this name was used already at birth "birth" or came only through "adoption" or "deed poll". There is considerable overlap with the *unmarried* name classifier and the other classifiers of Axis 2. Consequently we had to relax the notion that axis 2 classifiers need to be mutually exclusive.

Initials

We recognized the the term "initials" may have slightly different meanings in an international context. In the Netherlands "initials" are all the first letters of your given names and family names as you choose.

In Holland there is also the concept of *voorletters* which are the first letters of the given names. In Holland adults are normally recorded only using their voorletters and family names. This is similar to the vancouver citation style that never spells out first names.

However, we confirmed that the term "inital" means first letter (of whatever), regardless of given or family name. The beautiful initials that start a chapter of medieval books are called "initals" too (e.g., the Schwabacher initals). When "initals" is used in the plural form in context of names and signatures, it usually refers to all the initials of given and family names. It is then used as a short form of a signature.

A typical dutch name using only *voorletters* would be recorded as a person name variant. We would not need to associate initals with spelled-out name parts.

Academic titles

Academic titles and professional credentials are like voorveugsels and noblety titles on axis 4. You can classify academic degrees and professional titles as suffixes or prefixes. This keeps track of the problem that "PhD" and "MD" are suffixes but "Dr." and "Prof. Dr. med. Dr. phil. h.c." are prefixes.

3.5.4 Organization Name

We need much less flexibility and power with organization names. We considered what might be to organization names:

- Different name parts, such as "Hewlett-Packard" vs. "HP" vs. "Inc.", "Co.", "Ltd.", "B.V.", "AG", "GmbH", etc.

- "Marriage" of companies and trading of divisions, thus, UNIX was a trade mark of AT&T, then USL, then Novell, and who knows. "Daimler" and "Crysler" are now "Daimler-Crysler" and "Behring", a manufacturer of vaccines, is known or subsumed by some other name in the U.S.

Anyway, we concluded that noone really keeps track of those things, so all we need is an organization name string and, perhaps, a name type code. HL7 v2.3 had a name type code table for organization names (XON) including:

Organization Name Type Codes (adopted from HL7 v2.3)

L	legal
A	alias
D	display
ST	stock exchange

Display name has no defined use, since names are always displayed and it begs the question "whose display?". I wonder whether anyone in healthcare would want to include the Wall Street ticker symbol or the Indianapolis Star newspaper’s abbreviation of a manufacturer of vaccines. But there is no reason why we should restrict this existing "feature" of version 2.3.

All in all this is not a very controversial or important issue. So, unless there is any significant objection we can just stick to a v2.3-like solution.

Organization Name Variant (ON)			
A name for an organization. (What else is there to say?)			
component name	**type/domain**	**optionality**	**description**
type	Code Value [p. 58]	**optional**	A code indicating what an organization name is to be used for. Examples are: alias, legal, stock-exchange.
value	Character String [p. 38]	**mandatory**	The actual name data as a simple character string.

Note: this has changed. In a previous draft the Organization Name (ON) was a set of Organization Name Variants (ONXV) with no additional information. It is therefore simpler to define ON in parallel with PN as representing one name variant and let PAFM handle the rest in the RIM.

Note: a harmonization request to PAFM is required for the Organization class to

1. delete attribute: Organization.organization_name_type_cd

   Rationale: Attribute duplicates the ON.type component of the Organization name data type.

2. rename attribute: Organization.organization_nm to "nm"

   Rationale: Name does not conform to the MDF style guide as it repeats the name of its class.

3. assign data type: Organization.nm : SET<ON>

4 Quantities

4.1 Overview

All our quantitative concepts can be constructed by the means that mathmatics has developed during the past 3000 years. The most fundamental and abstract quantitative concept is the number. There are different kinds of numbers. Primarily there are natural numbers (1, 2, ...), cardinal numbers (0, 1, 2, ...) and integer numbers (..., -2, -1, 0, 1, 2, ...). Such numbers are the results of enumerating, counting or simple calculations (+, -, ·, ÷, mod) with integer numbers. The set of integer numbers is countably infinite and discrete.

Next there are rational numbers that are constructed through division (1/2, 1/3, 2/3, 1/4, ...). The set of rational numbers is continuous and infinite but still countable (G. Cantor). Geometry has introduced irrational numbers (e.g., square root of 2, pi, ...). The superset of rationals and irrational numbers is called real numbers. The set of real numbers is continuous, infinite, and not countable.

The ancient Arabs have introduced the custom to represent numbers as decimal digits where each position has a certain value. This Arabic numbering system was a great advance over the ancient Hebrew and Greek custom to use letters as numbers, or the arcane Roman number system. With Arabic numbers one could calculate much easier.

However, numbers with decimal point can only approaximate most rational and irrational numbers, hence, numbers with a decimal point can not be considered exact.

Most computer programming languages distingush between the two data types integer and real (floating point) number. Some know rationals and complex numbers. Whereas HL7 v2.x had only one data type for numbers, HL7 v3 will distinguish between interger and real. This distinction is suggested not just by technological considerations (both are implemented quite differently).

The main reason for distinguishing integer and real numbers is about semantics. Integer numbers are exact results of counting and enumerating. In natural science and real life, integer numbers are rather rare. Measurements, estimations, and many scientific computations have real numbers as their results, imprecise real numbers. Measurements are but approximations to the quantitative phenomena of nature.

There are other distingished quantitative phenomena that can be partially described by numbers but which have a meaning beyond numbers. Among such quantitative phenomena are physical measurements with units of measure, money, and real time as measured by clendars.

This specification defines data types for integer and real numbers, for physical measurements, money, and calendars. There are many more quantitative phenomena that we may or may not define data types for in the future. Examples for those we will define are vectors, waveforms, and possibly matrices. We will probably not consider complex numbers, except if a concrete use case appears.

4.2 Integer Number

Integer Number (INT)
Integer numbers are *precise* numbers that are results of counting and enumerating. Integer numbers are discrete, the set of integers is infinite but countable. No arbitrary limit is imposed on the range of integer numbers. Two special ineger values are defined for the positive and negative infinity.
PRIMITIVE TYPE

No fixed arbitrary limits on value range

No arbitrary limit is imposed on the range of integer numbers. Thus, theoretically, the capacity of any binary representation is exceeded, whether 16 bit, 32 bit, 64 bit, or 128 bit size. Domain committees should not limit the ranges of integers only to make sure the numbers fit into current data base technology. In finance and accounting those limits are frequently exceeded (e.g., consider the U.S. national budget expressed in Italian Lira or Japanese Yen.) Designers of Implementable Technology Specifications (ITS) should be aware of the possible capacity limits of their target technology.

The infinity of integer numbers is represented as a special value. The representation of integer numbers is up to the ITS. In our instance notation we use the special symbol `#iinf` for positive infinity ($Aleph_0$), `#niinf` for negative infinity (- $Aleph_0$.) Note that `#niinf` = - `#iinf`.

Constraints on value ranges

In cases where limits on the value range are suggested semantically by the application domain, the committees should specify those limits. For example, the number of prior patient visits is a non-negative integer including 0.

Although we do not yet have a formalism to express constraints, we should not hesitate to document those constraints informally. We will eventually define (or deploy) a constraint expression language.

ITS Presentation and Literals

We allow integer numbers to be represented by character string literals containing signs, decimal digits, and symbols for infinities. *Implementable Technology Specifications* (ITS) such as for XML will most likely use the string literal to represent integers. Other ITSs, such as for CORBA, might choose to represent integers by variable length bit strings or by choices of either a native integer format or a special long integer format.

We may even want to define non-decimal representations in bases 2, 8, 16, and 64.

4.3 Real Number (was: Floating Point Number)

Note: can we change the name in the last minute? I realized too late that calling it "Floating Point Number" is incorrect, since that name refers to a particular computer-representation of a number. I would now much rather call it "Real".

Real Number (was: Floating Point Number, FPN)			
A data type that approximates real numbers to a certain precision. Real numbers occur whenever quantities of the real world are measured or estimated or as the result of calculations that include other real numbers.			
component name	**type/domain**	**optionality**	**description**
value	*Real Number*	**required**	The value without the notion of precision or with an arbitrary precision. We do not specify a data type for true real numbers of infinite precision.
precision	Integer Number [p. 135]	**required**	The precision of the real number in terms of the number of significant decimal digits.

Semantic components vs. representational components

A real number has the semantic components *value* and *precision*, however, this does not necessarily mean that any representation of a floating point number will be a structure of two distinct components. Especially, since it is not possible to define a data type for true real numbers of infinite precision, the *value* component is not of an existing data type.

Rather than being components of the data type "value" and "precision" that can be evaluated on the application layer. These properties must be kept invariant throughout all ITS implementations. This is especially an issue if binary floating point numbers are used, such as IEEE 754.

Precision

The precision of a real number is defined here as the number of decimal digits. According to Robert S. Ledley [*Use of computers in biology and medicine*, New-York, 1965, p. 519ff]: "A number composed of *n* significant figures is said to be *correct to n significant figures* if its value is correct to within 1/2 unit in the least significant position. For example, if 9072 is correct to four significant figures, then it is understood that the number lies between 9072.5 and 9071.5 (that is 9072 ± 0.5) [...]"

Obviously this method of stating the uncertainty of a number is dependent on the number’s decimal representation. For binary representations we could, in principle, specify the precision more granularly. However, the statement that a value lies within a certain range is problematic anyway, because it begs the question about which level of confidence we assume. We will define a generic data type for probability distributions that allows exact statements of uncertainty.

Sometimes the term *precision* is put in opposition to *accuracy*. Where precision means the exactness of the numeric representation of a value, accuracy refers to the smallness of error in the measurement or estimation process. While those concepts can be distinguished, they are related inasmuch as we do not want to specify a higher precision of a number than we can justify by the accuracy of the measuring process generating the number. Conversely, we do not want to specify a number with less precision than justifiable by the accuracy.

In fact, there is considerable confusion around the meaning of such terms as precision, accuracy, error, etc. There is hardly a commonly accepted definition of those terms. A review of some of the available literature on that topic may help: the NIST’s Guidelines for the expression of uncertainty in measurement. (http://physics.nist.gov/cuu/Uncertainty/index.html) which in turn is based on the ISO’s *International Vocabulary of Basic and General Terms in Metrology* (VIM). In addition, the European standard ENV 12435 *Medical informatics - expression of the results of measurements in health sciences*, in its normative Annex D, summarizes the NIST’s position.

To summarize: NIST’s *Guidelines*, and ISO’s VIM regard the term *accuracy* as a "qualitative concept". Other related terms are *repeatability*, *reproducibility*, *error* (*random* and *systematic*), etc. All those slightly different but related and overlapping concepts have been subsumed under the broader concept of *uncertainty* in a 1981 publication by the International Committee for Weights and Measures (CIPM) in accordance with ISO and IEC. The uncertainty of measurement is given as a probability distribution around the true measurement value (measurand). Given such a probability distribution, a value range can be specified within which the true value is found with some *level of confidence*.

These concepts of specifying accuracy based on statistical methods are well known in the medical profession. However, these statistical methods are quite complex, and exact probability distributions are often unknown. Therefore, we want to keep those separate from a basic data type of real numbers. However, a data type for real numbers can only be an approximation to true real numbers and we want to account for this approximative nature by keeping a basic notion of precision in terms of significant digits right in the real number data type.

In many situations, significant digits are a sufficient estimate of the uncertainty, but even more important, we must account for significant digits at interfaces, especially when converting between different representations. For instance, we do not want a value 4.0 to become 3.999999999999999999 in such a conversion, as it happens sometimes when converting decimal representations to IEEE binary representations.

No fixed arbitrary limits on value range

No arbitrary limit is imposed on the range or precision of real numbers. Thus, theoretically, the capacity of any binary representation is exceeded, whether 32 bit, 64 bit, or 128 bit size. Domain committees should not limit the ranges and precision of real numbers only to make sure the numbers fit into current data base technology. Designers of Implementable Technology Specifications (ITS) should be aware of the possible capacity limits of their target technology.

The infinity of real numbers is represented as a special value. The representation of real numbers is up to the ITS. In our instance notation we use the special symbol `#finf` for positive infinity ($Aleph_1$), `#nfinf` for negative infinity (- $Aleph_1$.) Note that `#nfinf` = - `#finf`.

Constraints on value ranges

In cases where limits on the value range are suggested semantically by the application domain, the committees should specify those limits. For example, probabilities should be expressed in real numbers between 0 and 1.

Although we do not yet have a formalism to express constraints, we should not hesitate to document those constraints informally. We will eventually define (or deploy) a constraint expression language.

ITS Presentation and Literals

We allow real numbers to be represented by character string literals containing signs, decimal digits, a decimal point and exponents. An ITS for XML will most likely use the string literal to represent real numbers. Other ITSs, such as for CORBA, might choose to represent real numbers by variable length bit strings or by choices of either a native (IEEE 754) floating point format or a special long floating point format.

Decimal real numbers can be represented in a standard way, so that only significant digits appear. This standard representation always starts with an optional minus sign and the decimal point, followed by all significant digits of the mantissa followed by the exponent. Thus 123000 is represented as "`.123e6`" to mean $.123 \times 10^6$; 0.000123 is represented as "`.123e-3`" to mean $.123 \times 10^{-3}$; and -12.3 is represented as "`-.123e2`". to mean $-.123 \times 10^2$.

The reason why we define decimal literals for data types is to make the data human readable. To render the value 12.3 as "`.123e2`" is not considered intuitive. The European standard ENV 12435 recommends that the exponent should be adjusted such as to yield a mantissa between 0.1 and 1000. Those representations tend to be easier to memorize. The external representation is of the form:

sign	::=	**`+`** | **`-`**
digit	::=	**`0`** | **`1`** | **`2`** | **`3`** | **`4`** | **`5`** | **`6`** | **`7`** | **`8`** | **`9`**
digits	::=	*digit digits* | *digit*
decimal	::=	*digits* **`.`** *digits* | **`.`** *digits*
mantissa	::=	*sign decimal* | *decimal*
exponent	::=	*sign digits* | *digits*
real	::=	*mantissa* **`e`** *exponent* | *mantissa*

Number of significant digits

The number of significant digits is determined according to Ledley (ibid.) and ENV 12435:

1. All non-zero digits are significant.
2. Leading zeroes are not significant, regardless of the decimal point's position.
3. All trailing zeroes are significant, regardless of the decimal point's position.

Note that rule number 3 diverts from Ledley and ENV 12435. Judgment about the significance of trailing zeroes is often deferred to common sense. However, in a computer communication standard common sense is not a viable criterion (common sense is not available on computers.) Therefore we consider all trailing zeroes significant. For example 2000.0 would have five significant digits and 1.20 would have three. If the zeroes are only used to fix the decimal point (such as in 2000) but are not significant we require to use exponents in the representation: "`2e3`" to mean "2×10^{3}".

4.4 Ratio

HL7 v2.3 defined the data type "structured numeric" (SN) for various purposes. Among those purposes was to cater the need to express rational numbers that often occur as titers in laboratory medicine. A titer is the maximal dissolution at which an analyte can still be detected. Typical values of titers are: "1:32", "1:64", "1:128", etc. Powers of 1/2 or 1/10 are common. Sometimes titer results are falsely represented by writing donw only the denominator (e.g. 2 meaning 1:2 and 128 meaning 1:128). Great confusion exists in practice when comparing titers to reference values. Such, one almost always sees or hears statements like "1:256 > 1:128" when the opposite is true.

Regardless of how negligent those titers are commonly treated in medical praxis, titers are rational numbers. In the inroduction, however, we noted that rational numbers are exact. Titer values sure are measurements, and all measurements are inexact.

Thus, in theory, a titer of 1:128 could be reported as 0.0078125. However, no human user would understand such a result. One could recover the original ratio using the inverse of 10000000/78125 which is 128, but to do that, the receiver would have to know that the given number is to be presented to the user as a ratio of 1/*n*.

Since rational numbers are exact mathematical constructs, and since this exactness is not available in medicine, this specification defines a generalization of rational numbers, the Ratio. A ratio is any quotient of two quantities. Those can be two integers, in which case we have an exact rational number. But the quotient can be built as well from real number values, or physical measurements or any combination thereof.

Note that the ratio has the semantics of a quotient. The ratio data type must not be used only because it is a handy representation of two related values. Notably, blood pressure values, commonly reported as 120/80 mm Hg are **not** ratios!

Ratio (RTO)			
A ratio quantity is a quantity that comes about through division of a numerator quantity with a denominator quantity. Ratios occur in laboratory medicine as "titers", i.e., the maximal dissolutions at which an analyte can still be detected.			
component name	**type/domain**	**optionality**	**description**
numerator	*Quantity*	**required** default is 1	The numerator quantity.
denominator	*Quantity*	**required** must not be zero default is 1	The denominator quantity.

A Quantity is a generalization of the following data types:

- Integer Number [p. 135]
- Real Number [p. 136]
- PhysicalQantity [p. 141]
- MonetaryAmount [p. 142]
- Ratio [p. 139] (recursively)
- ... other quantitative data types

4.5 Measurements

4.5.1 Physical Quantities

All versions of HL7 v2.x had the data type "Composite Quantity with Unit" (CQ) defined. This data type, however, was not normally used in measurement observations (OBX). Instead, in an OBX you would send a numerical result (value type NM) and send the units in a separate OBX field. Moreover, units used to have different code tables depending on whether the CQ type or the OBX mechanism was used. We want to clean this up. It seems to be so natural to define a data type for measurements (or "dimensioned quantities") that many other standardization groups adopted (reinvented) this two component data type over and over again.

CEN TC251, WG 1, PT 26's first working document *Health Informatics; Electronic Healthcare Record Communication; Part 1: Extended Architecture* in table 25 [p. 52f] defines a type "quantity" as "A measurement expressed as a numeric value and unit of measurement" with the two component structure (value, unit).

The current draft 5 of CORBAmed's *Clinical Observation Access Service* (COAS) specifies an "MeasurementElement" that basically contains value and unit, however, the structure is slightly different.

We define the data type Physical Quantity as follows:

Physical Quantity (PQ)			
A physical measurement is a dimensioned quantity expressing the result of a measurement act. It consists of a value and a unit.			
component name	**type/domain**	**optionality**	**description**
value	Real Number [p. 136]	**required**	The magnitude of the quantity measured in terms of the unit.
unit	Concept Descriptor [p. 72]	**required**	The unit, which is a real world concept.

Units

Units are mathematical structures, quite different from other vocabularies. Armed with a little bit of mathematics, dealing with units is much simpler than dealing with the usual medical concepts. Units are hard to attack with semantic networks, but easy to deal with in simple algebraical structures. [More will follow, see also Schadow G, McDonald CJ, et al. *Units of Measures in Clinical Information Systems*. JAMIA. Apr/May 1999.]

Existing codes for units of measure are:

1. ISO 2955 (1983)
2. ANSI X3.50 (1986)
3. HL7 ISO+/ANSI+, equals ASTM 1238, equals HISPP MSDS CDT (based on ISO+).
4. There is a new *Unified Code for Units of Measures* (http://aurora.rg.iupui.edu/UCUM) (UCUM) that we will submit to either ANSI X3.50, ISO TC12, or as an HL7 defined code (probably maintained by Regenstrief, similar to LOINC). The UCUM is much more complete, does not suffer from ambiguities and imprecise semantics as the other codes do.

Regardless of what coding system HL7 ends up recommending (or mandating) we will be able to accommodate this in the above defined structure.

Constraints on the Dimension of a Measurement

Not all physical *kinds of quantities* (or *dimensions*) are applicable in every use of the measurement data type. Subsets of units of measures are defined through the semantics of units and could be specified in either of three ways:

1. with a special code for kinds of quantities,
2. with a special expression language (similar to the units code itself),
3. with a paradigmatic unit to which a given unit must be convertible.

Ad. 1: Examples for a special code for kinds of quantities is the "property" code of LOINC. I.e. "TIME" for time durations (e.g., seconds)

Ad. 2: Examples for a special expression language is the way dimensions are commonly specified, "T" for time, "L" for length, "LT^{-1}" for velocity, "LT^{-2}" for acceleration and "$LT^{-2}M$" for force.

Ad 3: If an attribute "encounter duration" is defined as a measurement then one could give the paradigmatic unit "s" (second) in the definition of that attribute, meaning that every value of this attribute must be convertible to seconds. This would be true for all measurements with units such as minute, hour, day, and many more.

4.5.2 Monetary Quantities: Currencies

Expressions of monetary amounts are of the same abstract form as physical quantities, i.e. a composite of a value and a unit (the currency unit). As with physical quantities, this composite can be regarded as a product (multiplication) of the value and the unit. As with physical units we have submultiples of currency units (e.g., dollar and cent, pound and penny, mark and pfennig, rupee and paisa, etc.) Currencies appear to be just another dimension of measured quantities.

However, there is also a big semantic difference between monetary units and physical units. While "exchange rates" of physical units are pretty stable over many decades, the value of monetary units is negotiated differently each day in different places of the world. While an international inch is 2.54 centimeters exactly (since 1959), a U.S. dollar (USD) may be 1.795 Deutsch mark (DEM) today and 1.659 DEM tomorrow. The same USD may be worth 1.795 DEM in New York and 1.801 DEM in Frankfurt (Germany) at the same time.

This suggests handling currencies differently from physical quantities. The methodology of this data type redesign work defines data types as semantic entities. The fact that some data types with different semantics may share a similar structures does not by itself warrant to lump both types together.

Monetary Amount (MO)			
A monetary amount is a quantity expressing the amount of of money in some currency.			
component name	**type/domain**	**optionality**	**description**
value	Real Number [p. 136]	**required**	The magnitude of the monetary amount in terms of the currency unit..
currency unit	Concept Descriptor [p. 72]	**required**	The currency unit (e.g., US$, Deutsch Mark, Pound sterling), which is a real world concept.

ISO 4217 is an international code for currency units. Although the standard text itself is copyrighted, the values themselves are freely usable and are listed here (http://www.triacom.com/archive/iso4217.en.html). This code does only cover the "major" currency units of each country, e.g. U.S. dollar but not cents, British pound but not penny, German mark, but not pfennig, Indian rupee but not paisa, etc. This shouldn’t be a major problem, since most currency submultiples are 1/100 worth the major unit (yes the British turned towards a decimal system as well, no "shilling" any more; was 1/16 pound sterling.)

Price Expressions

Expressions of monetary units and physical units may be mixed as in price expressions, such as 5 U.S. dollar (USD) per milliliter (price), or 20 USD per hour (salary). Two ways exist to construct price expressions.

1. using the Ratio [p. 139] data type with a monetary amount as numerator and a physical quantity [p. 141] as a denominator.
2. combining a code for physical units with a code for currency units.

(1.) The example price expressions above could be built with ratios as follows

```
(Ratio
  :numerator   (MonetaryAmount
                  :value 5.00
                  :currencyUnit "USD")
  :denominator (PhysicalQuantity
                  :value 1
                  :unit  "ml"))


(Ratio
  :numerator   (MonetaryAmount
                  :value 20.00
                  :currencyUnit "USD")
  :denominator (PhysicalQuantity
                  :value 1
                  :unit  "hr"))
```

This is a clean and the most simplest solution, since separate codes for physical units and currency units are available today. This allows to flexibly combine quantities that have different semantic properties.

The alternative (2.) is to merge a code for physical units with another code for currency units. This endeavor raises problematic questions about the differences in semantics.

The way this could work in UCUM is that one would define an eighth base unit in addition to the seven existing base units. This would probably be the U.S. Dollar, or one troy ounce of gold - traditionally used as the standard currency by the World Monetray Fund.

Lexically, the currency units would be treated just like any other unit. Semantically, however, their value would be taken from a dynamic table, which could be an on-line connecting directly to New York’s Wall Street or any bank institution regarded as authoritative in any given realm.

However, this raises question what happens if a message crosses a given realm? While conversions between physical units should be enabled because physical units of the same dimension are equivalent, currency units are not equivalent. Currency units do change their exchange rates on an hourly basis. While it does not matter at all whether you have 1 yard or 0.9144 meter, it does matter a lot whether you have 100 US Dollars or 3000 Indian Rupees.

This matter must be considered an open issue for the time being.

4.5.3 Things as Pseudo Units

Sometimes all kinds of things are used in expressions of the same form as physical quantities, such as

number × *unit*

Those expressions are often used when numbers are reported that are the results of counting things. For instance, if we count tablets and the number of tablets is 50, people naturally say, "50 tablets", which almost lets "tablet" appear as if it wehere a physical unit. However that is not true.

Not any object is a phyiscal unit. Moreover, the connection between things and physical units is mainly suggeted by European natural languages, where we say "50 tablets", "20 cars", "1000 chicken" and the like. Other languages, like Japanese, use category suffixes behind count numbers, such "5 pencils" would be "empitsu go-hong" in Japanese, where "hong" is used for all kinds of long and thin things. Should we therefore suggest to regard "hong" as a physical unit?

Those thing-units do have certain properties in common with physical units, for example, you can not add meters and seconds or apples and oranges. But there are also important differences. All international standards on measurements state that when object counts are reported, the measurement *name* should contain the things counted. One should not make up ad-hoc units. In lab data bases one frequently finds units such as "red blood cells" vs. "white blood cells", which is redundant, given that the measurement name is reported properly.

Those thing-unit are most common in the pharmacy, where they appear as medication units of application (e.g. tablet, capsule, vial, spray, etc.) that are often used as if those were units of measure. Those symbols, however, are **not** units of measure, because they are not inherently quantities. While a metre is inherently a quantity (worth approx. 3.4 foot), a tablet or vial has no magnitude by itself. A given tablet, vial or spray may have properties, such as strength or volume, but those are different for any different kind of tablet, vial or spray under consideration. Conversely, a metre does not *have* different quantitative properties, a metre *is* a quantity in essence. Tablet, vial, or spray are *not* essentially quantitative items.

Of course, you can count tablets (like you can count all kinds of things); of course, a tablet, as a physical body does have volume, length, width, and depth. But the essence of a tablet is its *form* and not any specific kind of quantity. Conversely the essence of a meter is a certain amount of length, the essence of a second is a certain amount of time, and the essence of a dollar is a certain amount of money. Not every kind of an object is a candidate unit.

One may argue that not even all units or measure are real units, so why should one bother? For example, international Units (i.U.) are units that do not have a fixed magnitude associated with them.

International Units are arbitrary units defined for every analyte by some international organization IUPAC (?). Examples are i.U. for penicillin, insulin, streptokinase, urokinase, and other medications, but i.U. are defined for many enzymes, hormones and antibodies. The rationale for those units is twofold:

1. these are functional units that measure a certain biochemical function rather than a specific molecule, because many slightly different molecules can carry out to the same biochemical function;
2. the measurement process has so many parameters which all need to be standardized that it is not possible to come up with comparable units, standardized across all analytes.

The units U (= 1 µmol/min) and katal (= 1 mol/s) of catalytic activity try to be standardized for all enzymes. However, the measurement conditions still need to be standardized because 1 katal of Phosphofructokinase measured at pH 7.4, 37 degree Celsius, in a Ringer solution, with this much ADP and no 1,2-Bisphosphoglycerate present, is quite different from 1 katal of the same analyte measured at pH 7.5, 28 degree Celsius, in plain water with only that much ADP present.

The various international Units (i.U.) are still essentially quantitative concepts, because international Units are defined for no other purpose than to measure quantities. This is quite different with tablets, vials, and sprays.

The order/results committee will have to work out the specifics on the relationship between units of application and units of measures in its information model. It is quite important for a clean information analysis to distinguish the semantics of physical units from those thing-units. An important purpose of this data type redesign is to facilitate information analysis, not to obscure it.

Therefore we abstained from defining any data type to support thing-units.

4.6 Time

The treatment of dates and times has always been somewhat of a sticky issue in most data type specifications. The problem being that humans usually keep time using calendars which are traditional and quite complex numerical and ordinal constructs. The western world today uses the Gregorian calendar consistently and keeps time using the Universal Coordinated Time system. However, as the developed post-industrial western world becomes more and more aware of inter-cultural issues, other calendar systems than the Gregorian calendar are increasingly recognized. For example, Java has adopted a separation between a data type for points in time (the class java.util.Date (http://aurora.rg.iupui.edu/doc/Java/docs/api/java.util.Date.html#_top_)) and an abstract type for various calendars (java.util.Calendar (http://aurora.rg.iupui.edu/doc/Java/docs/api/java.util.Calendar.html#_top_),) of which the Gregorian/Julian calendar was first implemented (java.util.GregorianCalendar (http://aurora.rg.iupui.edu/doc/Java/docs/api/java.util.GregorianCalendar.html#_top_)) IBM (http://alphaworks.ibm.com/tech/calendars) has made additional calendar classes available for Buddhist, Hebrew, Islamic and Japanese Imperial calendars.

While we believe that the western calendar will prevail to be the domiant calendar in healthcare we want to keep the separation of an abstract concept of time from calendar dependent time notations. While we do not specify support for any other than the Gregorian calendar in this document, we assume that local HL7 user groups can do so in a compatible way. This specification clearly distinguishes the following concepts related to time:

Point in Time [p. 148]

One specific point on the real time axis.

Interval [p. 166] of Point in Time [p. 148]

A continuous period of time between one start and one end time. Start and end times may be infinite or unspecified, which allow the expressions of e.g. open ended periods.

Duration of Time [p. 148]

A duration is a quantity of time with no particular start and end time just as a length is a quantity without a particular place in space. Durations of time are nothing else than a Physical Quantity in the dimension of time.

Periodic Points or Intervals of Time [p. 153]

There is a number of time expressions we use to specify periodic events. Those may be as simple as "every day at 08:00" or as complex as "the second Sunday of the month May."

Arbitrary sets of Time [p. 170]

In scheduling recurrent events one may want to build arbitrary sets of points in time or sets of intervals of time, recurrent in possibly very complex patterns. Not yet supported in all generality.

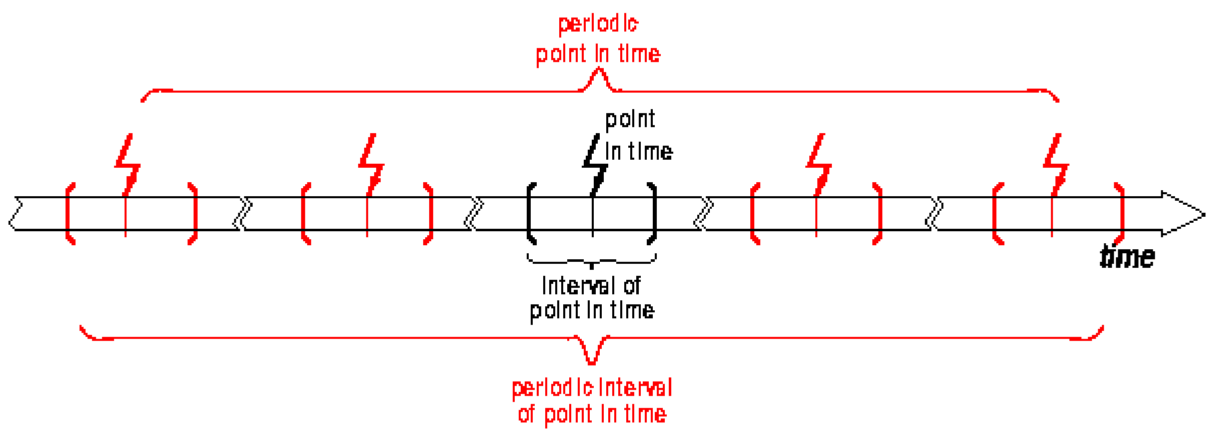


Figure 6: Time-related phenomena recognized by this data type specification.

We do not consider event- or activity-related concepts as in the scope of time. For example, such expressions as "at the hour of sleep" or "before meal" or "since a particular accident" are not genuinely concepts of time. Events and activities are naturally related to time, but are still quite different from time.

4.6.1 Time Durations

Some recently developed type systems define a special data type for durations (e.g. for instance the one developed by M. Stonebreaker for the POSTGRES object-relational data base project) The Arden syntax also knows such a concept. In this v3 data type model, however, time durations are but a special case of a physical quantity. Durations of time are nothing else than measurements in the dimension of time. Thus those durations have the units 1 s, 1 min, 1 hr, 1 d, 1 wk, 1 mo, 1 a, etc.

4.6.2 Point in Time

Point in Time (TS) (also called "time stamp")
A point in time is a scalar defining a point on axis of natural time. This naive concept of an absolute time scale is not concerned with relativity of time as is important in astrophysics and cosmology.
PRIMITIVE TYPE [see text]

We conceptualize time in a naive sense as one universal continuous and even dimension of time, just as in a fourth Euclidean dimension in our coordinate system of events (space and time.) This notion of time is naive in the same sense as our typical notion of space is naive given the Relativity theory. However, this notion reasonably approximates "reality" for all purposes of earthly health care.

This natural time scale is, almost like the temperature scales (Celsius or Fahrenheit), an *interval scale* (aka. *difference scale*). Such interval scales are characterized by an arbitrary choice of the origin (zero-point.) While the Celsius temperature scale defines a zero point at the freezing point of water and a standard degree as 1/100 of the boiling point of water, the modern western calendar (Gregorian calendar combined with the Universal Cooordinated Time) defines the zero point at the birth of Christ and the basic unit of time as the second. There are obvious problems with the determination of the zero point of the Christian calendar, and there have been really two Christian calendars (Julian and Gregorian,) but the principle is the same.

Zero points on the natural time axis are chosen arbitrarily, and called the "epoch".

Many data type specifications for point in time are based on an epoch. Examples for epochs are: 1/1/1970 00:00:00 UCT on Unix, 1/1/1980 00:00:00 UCT on MS DOS, 12/31/1959 00:00:00 EST in the Regenstrief MRS, 10/15/1582 00:00:00 UCT in CORBA’s COAS. Typical granularities are seconds, milliseconds, microseconds, or nanoseconds measured from that epoch. This way of representing time is very simple: all that is needed is a counter that counts the ticks of a clock since the epoch. Although this is not easily human readable, it is very easy to compute with those standardized time values.

Translations between epoch-granularity-counter systems (clocks) are simple linear translations between coordinate systems.

Even though clocks are based on some granularity, one can conceptually base a clock on a scale based on real numbers, so that theoretically the time is measured continuously in any unit of elapsed time from the epoch. For example, given an epoch of January 1 1996, one can specify points in time such as July 9 1999 2:45 PM simply as 30878.75 days. Obviously the granularity is unbounded, that is, given a precise measuring method one can specify the time exact to the millisecond, nanosecond, picosecond, and more. In the common floating point registers for real numbers on computers, the precision is reduced with greater distance from zero, which is just what one would expect, given that the epoche is reasonably near the present time.

So, a representation of time based on an epoch and a Physical Quantity would be all that is needed. Indeed, since this representation of time comes closest to our conceptulaization of real time, an Implementable Technology Specification (ITS) may choose such a representation for time. This ITS-independent data type specification, however, concerns itself with time representation only for two reasons:

1. To define literals used in examples and constraints, and electively used by an ITS.
2. To understand and account for the fact that time expressions are often aligned to calendars.

Obviously the epoch-duration form of a point in time value is not very useful for a literal expression of time. Though astronomers use a simple counting of days elapsed since noon, Monday, January 1, 4713 B.C. on the Julian calendar, in the health care world we are not used to look at time in this way. So, our literal expression of time must be based on a our calendar.

Calendars

Traditionally the even flow of time is "convoluted" in many cycles defined by calendars. Such cycles are years, months, days, hours, minutes, seconds. Those cycles are not synchronized. Traditionally calendars have been defined based on astronomical phenomena, however, calendar years, months and days are not attached directly to astronomical phenomena. The closest fit is the calendar day to the solar day, but the calendar month is definitely not the same as a lunar (synodal) month.

Figure 7 below visualizes a calendar as a trajectory summed up from four such cyclical movements, year, month, day and hour. Imagine a clock that measures those cycle, but where the pointers are not all stacked on a common axis but each pointer is attached to the end of the pointer measuring the next larger cycle.

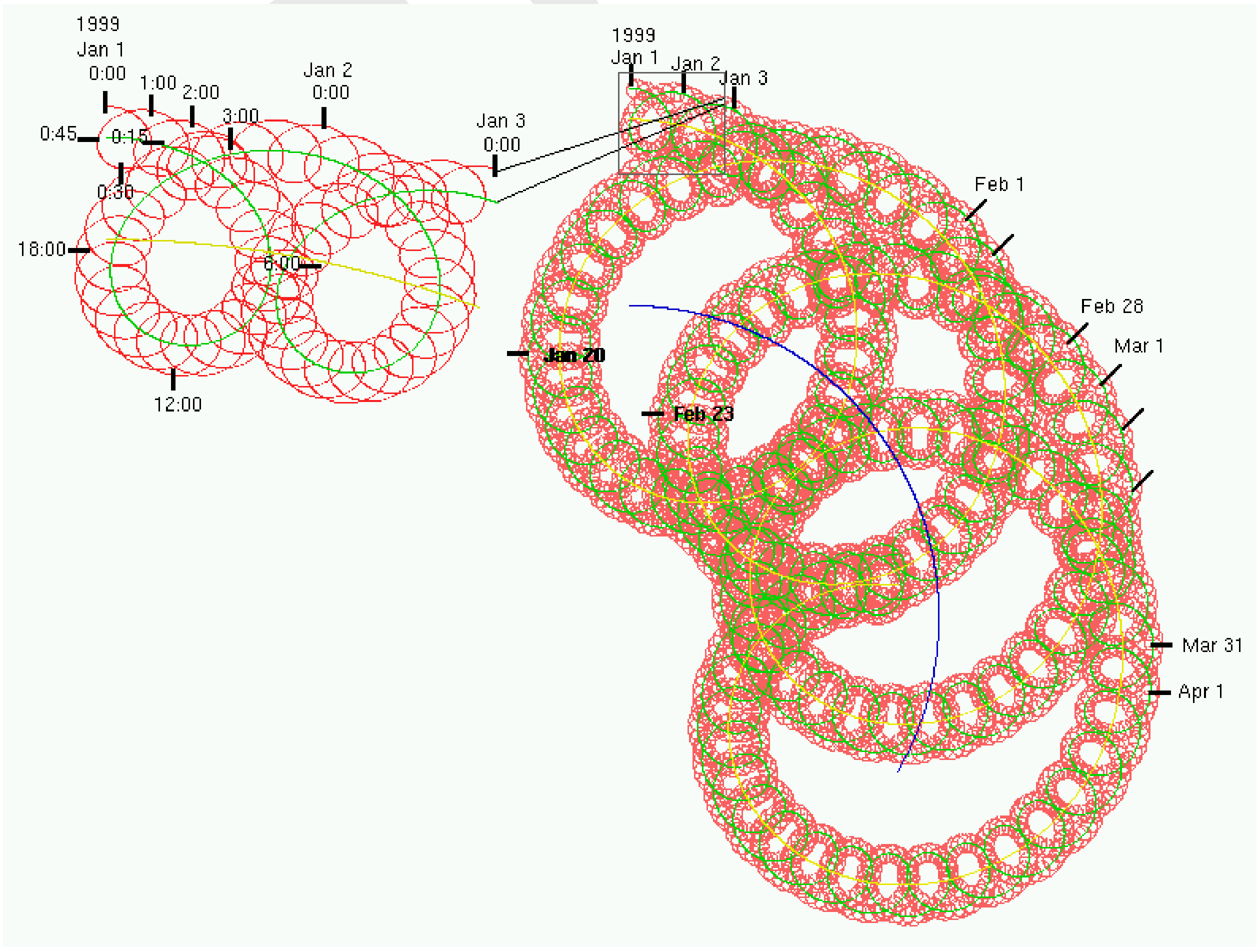


Figure 7: A calendar "rolls" the time axis into a complex convolute according to the calendar periods year (blue), month (yellow), day (green), hour (red), etc. The cycles need not be aligned, for example, the week (not shown) is not aligned to the month.

After rolling the time axes into the traditional cycles, a calendar expresses time as a sequence of integer counts of cycles, e.g., for year, month, day, hour, etc.

Because of the complex and often uneven relationship between the cycles, it is quite difficult to convert a calendar expression into an epoch/duration form. There are not just leap days (Feb. 29) added to leap years, but also leap seconds (added to leap days). The algorithms to determine leaps is difficult (leap year) or non-existent (leap seconds.) Leap seconds, for example, are determined sporadic and published as tables in Astronomical Almanacs. But fortunately, conversion is done by most operating systems, or by other available modules, such as the Java core library.

Literal Expressions for Point in Time

Quite solid standards for literal expressions of points in time based on the western calendar are HL7 v2.3's TS data type, and ISO 8601 (adopted in Europe as EN 28601). ASN.1's (ISO 8824) `GeneralizedTime` is a restricted form of ISO 8601. HL7's TS format is used by ASTM 1238 as well and lives on in ANSI HISPP MSDS CDT's `DateTime` format. Although HL7's TS format and ISO 8601 are similar, they also have considerable differences.

For HL7 v3 it seems worthwhile to consider adopting ISO 8601 [more about ISO 8601 (http://www.cl.cam.ac.uk/~mgk25/iso-time.html)]. However, ISO 8601 has some "features" that may be considered a disadvantage. First of all, ISO 8601 has too many unnecessary alternatives. A somewhat canonical date/time form is

YYYY-*MM*-*DD***T***hh***:***mm***:***ss*

The dashes between the date components, the colons between the time components and the "**T**" between date and time components may, according to ISO 8601, as well be omitted. The omission of those characters brings about a form very similar to ASN.1 or HL7's TS. The way of handling precisions in TS of HL7 v2.3 (after v2.2) is to leave out the less significant digits as required. However, without the "**T**" between date and time, this would be ambiguous with certain other valid ISO 8601 forms. ISO 8601 allows omission of the "**T**" by mutual agreement and only if no ambiguities are introduced - a clause that is usually hard to enforce (and therefore harmful) in standards.

The W3C is considering a subset of ISO 8601 (http://www.w3.org/TR/NOTE-datetime) for adoption. W3C's subset requires the "**T**" between date and time. The W3C schema working group, however, is using full featured ISO 8601 with all options.

Useful features of ISO 8601 that are not part of HL7's TS type are so called "ordinal dates" of the form

YYYY-*DDD*

YYYY-**W***ww*

YYYY-**W***ww-D*

These allow to specify a date as (1) the day of a year, (2) the week of a year, or (3) the week of the year plus the day of the week.

Moreover, ISO 8601 allows omission of more significant components (the delimiter dash, colon, or "**T**" must occur in those cases). This changes the semantics of the expression from a point in time to a periodic point in time. For example "---2" means every Tuesday, but subtle variations may have big impact on the meaning: "-W-2" means Tuesday "of the current week," which is a relative point in time.

Both, HL7’s TS and ISO 8601 handle time zones through offsets of the form "+hh:mm" or "-hh:mm" relative to UTC. TS adds a "Z" in front of the time zone suffix, while ISO 8601 uses the "Z" to mean UTC specifically (thus in ISO 8601 an offset expression following the Z would be contradictory).

Other worth-having features are missing in ISO 8601, however. Those missing features include the concept of significant digits available in TS, where you can say "198" to mean any year from 1975 to 1985.

It seems justified for HL7 to sticks with its own tradition of the TS data type. However, some slight changes could be applied to render most TS expressions compatible with ISO 8601 expression. Notably the "Z" should be used in the ISO 8601 style (i.e. only for UTC).

Furthermore the HL7 v2 TS format is very uniform and concise, which makes it suitable to be used as a model for literal expressions of other calendar times. Any new calendar to be defined needs to specify the calendar cycles, and their position and number of digits in the literal expression. In order to disambiguate literals from different calendars, the literal needs to be tagged by a calendar type code. This calendar type code will be prepended with a colon. The calendar type code for the western (Gregorian) calendar is "GREG" and need not be mentioned since it is the default.

4.6.3 Time Interval

A time interval is the continuous and uncountably infinite set of time points between a low bound and a high bound. The time interval is defined using the generic Interval [p. 166] data type defined further below. Low or high boundary can in principle be infinite but there is rarely any case where we need an infinite boundary, rather a boundary may be unknown, but is known to be finite. An interval can be specified incompletely by having an unspecified low or high boundary. For example, an Employment has a start date but need not have a fixed end date yet, in which case the high boundary is left unspecified. An interval also has a width which is the difference between high and low boundary. The width of the interval can be specified independently if neither the high boundary nor the low boundary is fixed.

The literal expression of an interval of time is defined by the generic literal expression for interval and point in time. Examples are:

`19980309-20000308`	March 9 1998 to March 8 2000
`<=20000308`	until March 8 2000 with unspecified begin date
`>=19980309`	from March 9 1998 with unspecified end date
`[26.58 h]`	unspecified boundaries but width of 26 hours 35 minutes is known

The width of an interval is specified using the data type for the difference of two points in time, which is a duration, and which in turn is expressed as a physical quantity.

4.6.4 Periodic Time

There are a number of problems that can neither be explained by a duration, nor a point in time, nor one specific time interval. The following are important examples in HL7:

- Office hours, for example:

Monday to Wednesday	08:00 to 16:00,
Thursday	08:00 to 21:00,
Friday	08:00 to 12:00,
closed on holidays	.

- "Snowbirds," e.g., people who periodically change addresses. For example, who live in Minneapolis between May 15 and September 15 and in Phoenix, Arizona from September 15 to May 15.
- Phone numbers to use on workdays from 08:00 to 17:00 and another phone number for evenings, weekends, and holidays.
- Medication schedules. For example, Amoxicillin 3 times a day for 10 days, or Cumarin 1 tablet Monday and Friday, and 1/2 tablet on Wednesday and on Sunday.
- Other schedules, e.g., home health care visit every other day in the morning, and every second saturday in the afternoon.

The industry has developed nifty ways to specify these phenomena based on the western calendar. While this is intuitive for humans, it can not easyly be translated between calendars. Therefore we invest some effort here looking for a conceptualization of these phenomena that is independent from a particular calendar as much as possible.

Periodic phenomena in physics are likened to waves which in turn are described by rotations. The rate of rotations is described either by frequency *f* (full rotations per time,) period duration *T*, or angular velocity *omega*.

In addition to the rate of rotation, there is an offset angle *phi*, called phase. The concept of phase becomes clear if you imagine two wheels rotating with the same frequency but the first wheel has started rotating a short time before the second wheel. Thus, the first wheel is always ahead of the second wheels position by some rotational angle. We can measure phase in either the angle or in the amount of time elapsed between the start of the first wheel and the second wheel.

Many periodic events are most naturally specified with a frequency. For example, in "amoxicillin 3 times a day" the periodic event "give amoxicillin" is timed by a frequency *f* = 3 /d. A frequency can be interpreted in two ways: either as an exact timing of the recurring event, such that it is distributed evenly, or as a statistical timing, such that the individual events occur at variable intervals but at average occur at the specified frequency. The latter is the typical case in medication scheduling.

If the intervals between the recurrences are not even, and if they need to be specified precisely the frequency and phase is not enough. For example if we want the 3 events per day scheduled at 7:00, 11:30, 17:00, we use the *time of day* to specify when the event is to occur. But what exactly is *time of day*, or the similar phenomena, such as *day of the week*, *month of the year* or *week of year*? Obviously those expressions are closely related with the cycles of the calendar.

On a simple digital calendar expression according to the HL7 TS format, such as "199907091956", we intuitively know what time of day and the like is: those expressions come about if we delete the high-order digits from the left. For example "*yyyymmdd*1956" stands for some day at 7:56 PM.

Since a calendar divides the even flow of time into cycles and counts full cycles in integer number, we are reminded of congruences, modulus and remainders.

A modulus is the remainder of an integer division. For example, 12 modulo 7 is 5. A congruence is similar to an equation based on an equality operation that partitions the set of integer numbers into remainder classes. For example 5 = 12 (mod 7), but also 12 = 19 (mod 7). Such a congruence is like the integer variation of a rotation described above, where the modulus (e.g., 7) is the period and the remainder is the phase.

If we have the time defined as epoch + duration in days, we can tell the day of the week of any date if we know the day of the week of the epoch. For instance, let our epoch be Monday January 1 of 1996. We can easily tell the weekday of July 9 1999. The duration between the epoche and the example date is 1285 days. Since the week is a cycle with period 7 days, we take 1285 mod 7 = 4. This is for Monday = 0, Tuesday = 1, ..., Friday = 4, a Friday.

Other such congruences can be constructed, all of which have the form:

remainder	of the	*modulus*
day	of the	week
month	of the	year
day	of the	month
week	of the	year
day	of the	year
hour	of the	day
minute	of the	hour
second	of the	minute

All those units are defined by the calendar and aligned to the calendar and thus are different from related units defined as averages for time durations. For instance, the average Julian month is 30.4375 days, but a calendar month varies between 28 and 31 days. Thus the congruence expression "month of the year" must be made available by the calendar and can not easily be calculated using the average month.

We can form more complex congruence expressions that are not provided by the calendar. For example, "every other Tuesday" is described by the congruence

days since epoch = 1 (mod 2 x 7); given that 0=Monday, 1=Tuesday, ...

while every Tuesday would be:

days since epoch = 1 (mod 7); given that 0=Monday, 1=Tuesday, ...

The congruences are a first step towards understanding what happens when we delete digits from the right of our calendar time expressions. Since the calendars are constructs of integer numbers, congruences are the way to express the calendar cycles. The disadvantage of this is that with one such modulus expression we can only express a recurring *points* in one such cycle. What if we want to specify recurring *ranges*, say, something like: every Monday from 9:00 to 15:00?

We can use a period and a phase, instead of a modulus and a remainder. In fact, periodic continuous functions in analysis are the counterpart of congruences in number theory. Both, period and phase can be measured in elapsed time, and are not restricted to integers. Monday 0:00 would be:

period = 1 week
phase = 0 days

assuming that our week-cycles start at a Monday. In order to get to Monday at 9:00 we need to increase our phase slightly by 3/8 = 0.375 days. Thus Monday 9:00 is

period = 1 week
phase = 0.375 days

Thus we can move the phase within the one week period to any time we like. For example, Thursday 12:00 noon would be

period = 1 week
phase = 3.5 days

This specifies a periodic point in time. If we want a periodic interval of time, such as Mondays 9:00 to 15:00, we can specify the phase as an interval:

period = 1 week
phase = [0.375; 0.625] days

This works for other cycles just as well. For example, the 5th day of the month 0:00 is given as

period = 1 month
phase = 5 days

And the entire 5th day of the month from 0:00 to 23:59:59.9999... is given using the right-open interval [5; 6[

period = 1 month
phase = [5; 6[days

The time of day 8:00 to 9:00 is given as

period = 1 day
phase = [8; 9[hours

and the entire February, the second month of the year, is given as

period = 1 year
phase = [1; 2[month

because we count from January = 0, February = 1.

Note that the ordinal numbers of the months and the weekdays depend on the selection of the epoch. A particularly useful epoch is January 1 1996 0:00 if one wants to start counting months and weekdays from 0-11 and 0-6 respectively, since January 1 1996 is a Monday.

The Snowbird's address that is different between April 15 and September 15 than over the Winter, can be expressed as

period = 1 year
phase = [3.5; 8.5[month
--> Minneapolis

period = 1 year
phase = [8.5; 3.5[month
--> Arizona

The telephone numbers being different from 9:30 to 19:45 (office) and from 20:30 to 9:00 (home) can be expressed as:

period = 1 day
phase = [9.5; 19.75] hour
--> office: 1 317 630 7960

period = 1 day
phase = [20.5; 9] month
--> home: 1 317 816 0516

Note that the bounds of the periodic interval need not be in order, since the period is cyclic.

This method can explain what a "time of day" and a "day of the week" are, independent from any particular calendar. The challenge of these period/phase expressions, however, is that the calendar units of year, month, day, hour, and minute are variable due to leap years, 28-29-30-31 days per month, and due to UCT leap seconds. So, the timing needs to be aligned to our calendar "grid", which makes the evaluation of those expressions subject to the same difficulty that we find translating a time given as epoch-duration into a calendar expression. In the example "5th day of every month", the period 1 month differs in duration from month to month, so, to calculate a series of points in time in a epoch-duration form, the calendar must be repeatedly asked for how many days are in any given cycle.

Periodic Times as Sets

More complex periodic times can be expressed based on the simple period/phase model. This is easy if we realize that periodic points in time and periodic intervals of time are special kinds of sets. Those sets are not continuous, meaing that they have "holes" between the repreating points or intervals of times. Also, those sets are infinite, as the periodic time will be defined along the entire time axis from prehistoric past to distant future.

We can combine sets using the operations for union ($\cup$) and intersection ($\cap$) to form the complex specification of business hours shown in the introductory example:

Monday to Wednesday	08:00 to 16:00,
Thursday	08:00 to 21:00,
Friday	08:00 to 12:00,

((period = 1 week; phase = [0; 3[day) $\cap$ (period = 1 day; phase = [8; 4] hour)) $\cup$
((period = 1 week; phase = [3; 4[day) $\cap$ (period = 1 day; phase = [8; 21] hour)) $\cup$
((period = 1 week; phase = [4; 5[day) $\cap$ (period = 1 day; phase = [8; 12] hour))

If we were open every Second saturday from 8:30 to 14:00 we could "add" the term:

$\cup$ ((period = 2 week; phase = [6; 7[day) $\cap$ (period = 1 day; phase = [8.5; 14] hour))

A few special cases are quite hard to express in our simple period/phase forms. For example, take another Snowbird who is out of Arizona between April 15 and October 15 and consider we need to be exact to the minute with this timing. While April has 30 days and thus April 15 cuts the month into half, the phase for April 30 is 3.5 month, which starts the Minneapolis season in the early morning of April 15 at 0:00 midnight. For October 15, we would like to use the phase 9.5, however this yields October 15 at 12:00 noon as the end of the Minneapolis period. For February 15 a phase of 1.5 would come out one entire day early February 14, 0:00 midnight on non-leap years and 12:00 noon on leap years.

The problem is that the calendar month is not constant, so that we are sometimes off by a few days if we choose the month as the phase of our periodic time expressions. If we chose the day of the year, we could average our error of one day over the entire year, but we would not be exact to the hour let even to the minute. We could concatenate our periodic time expressions, however, to successivly narrow down the exact start and end of the period:

((period = 1 year; phase = [104; 288[day) $\cap$ (period = 1 month; phase = [14; 15[day) $\cup$
(period = 1 year; phase = [105; 287[day))

This means, we first select an approximate range starting at April 15 on non-leap years and April 14 on leap years, and ending at October 16 on non-leap years and October 14 on leap years. That way we know that our maximal range is covered every year. Then we cut out the precise day, which is the entire 15th day every month (starting at 0.) This cuts out all times from the range but the 15th day of every month from April to October. The final expression "adds" the range from April 16 to October 15 (non-leap years) and April 15 to October 14 (leap years) back in.

An other case where our period/phase expressions are difficult to use is for Mother's day, i.e. the second Sunday in May. Since the week cycle is not aligned to the month and year cycle, we have a hard time to figure out the second Saturday of a month. The following period/phase expression describes Mother's day:

((period = 1 year; phase = [4; 5[month) $\cap$ (period = 1 month; phase = [7; 14[day) $\cap$ (period = 1 week; phase = [6; 7[day))

First we set the scope into the entire month of May (Jan = 0, ..., May = 4), then we narrowed into the second 7 day period, under the assumption that the second Saturday of the Month will always fall in the second 7-day period. Finally we take the period to be the week cycle and choose Sunday. This will select the second Sunday in May.

The last two examples show that our period/phase expressions are sometimes quite difficult to use. Note that our method did not fail in these examples, it was only difficult to apply. However, as of yet we have no mathmatically strong proof that there could never be an expression in some calendar that our method could not cover using a finite term of period/phase expressions concatenated by union and intersection operations.

Literal Expressions for Periodic Times

The period/phase expressions for periodic times are conceptualizations of periodic times that are independent from any representation of calendar dates. Given that such periodic times are supposed to be aligned to some calendar, our conceptulaization is still independent from any special calendar, and thus can be used with our Gregorian calendar as well as with the Hebrew, Islamic, Japanese Imperial, or other calendars.

Alas, these expressions are not suited to be shown or entered by humans and require considerable thought to be implemented correctly. Therefore a good literal expression is even more important for periodic times as it is for simple points in time.

Most of the nifty methods the industry came up with to specify the timing of periodic events are based on some representation (not on an abstract model such as our period/phase congruences.) Yet there seems to be no one representation that is widely accepted as a standard. [This statement is informed by a non-extensive research of the topic on the Internet. The most prominent such standards seems to be the UNIX "crontab" file format, and the work by the IETF Working Group on *Calendaring and Scheduling*, RFC 2445 "Recurrence Rule".]

It is therefore justified to craft a representation for periodic times for HL7, that is not based on any one such standard. HL7 has its own legacy that has proven useful. Especially the time stamp (TS) literal representation form for points in time has its merits to be simpler and more uniform than the many optional forms suggested by ISO 8601.

Our representation of periodic times will be based on the literal form of points in time. The general approach is that one can use a TS pattern to specify points in time conforming to a certain set of periodic points in time. The simplest such pattern being the removal of digits from the left to yield forms similar to the old HL7 TM data type. Nevertheless we will take ideas from existing specifications where appropriate.

For example, where "199907121555" is a point in time (precise to the minute), "1555" is just the time of day "component" of that time. This maps to the period = 1 day, phase = 15.916.

Note that the expression "1550" could as well be an TS precise to the year, since there is no way to distinguish what the value of the digits are. So, if we allow for leftmost digits to disappear, we need to either replace them by some other character to fix the position of the remaining digits (e.g., "########1550",) or we need to tag the remaining digits to indicate what they are (e.g., "H1550", where "H" stands for *hour*, the value of the following pair of digits.)

The crontab file format also defines periodic points in time using a pattern approach on the calendar and crontab uses a positional identification of the meaning of the components. For each of the calendar cycles: minute of hour, hour of day, day of month, month of year, and day of week, crontab allows to set individual values, repeat ranges, and step values.

Th HL7 TQ data type on the other hand, used a tagged approach to say "Q2H" for every two hours. However, this did not allow to specify the 55th minute of every hour.

The following is the EBNF specification of the literal expression for periodic points and intervals of time:

```
expr    := term [ + term ]
term    := factor [ [ ’ ’ ] factor ]
factor  := period [ period ] [ list |  [ range [ step ] | step ] ]
period  := <period identifier>
list    := number [ ’,’ list ];
range   := ’*’ | number ’-’ number;
step    := ’/’ number [ ’%’ number ];
number  := <number format depending on period>
```

A period identifier is a short one or two letter code for a calendar cycle. Period identifier come in three forms: (1) continuous, (2) ordinal, and (3) implicit. A continuous period is measured from some initial date (e.g., the epoch or an order start date) and is not bound to the larger calendar cycles. For example, if something is to happen strictly every other day regardlesss whether months are 30 or 31 days long one would use a continuous period. Continuous periods are formed using the letter `C` before the period identifier (e.g., `CD` for continuous day.)

An ordinal period identifier is aligned to the larger calendar cycles. For example, if something is to happen on every odd day of the month (1, 3, 5, ..., 27, 29, (31)) an ordinal period is used. Ordinal periods are specified using two period identifiers, one for the period in which to count and another for the larger period which we want to align to, (e.g. `DM` ordinal day of the month, `DW` ordinal day of the week.) Ordinal periods are counted from either 0 or 1 depending on the customs of the calendar. For example, in the western calendar day of the month and month of the year is usually counted from 1, while hour of the day and minute of the hour is counted from 0.

Implicit periods are those periods identified by the one letter period code, because it is so common to use it in either the continuous or the ordinal sense. For example, the year is counted continuously because there is no larger cycle (except for decimale multiples *decade* and *century*, which are not real calendar cycles.) Weeks are usually counted in a continuous way (i.e. not aligned to the calendar year,) while most other calendar cycles are aligned to each other (month-day-hour-minute-second.)

Period Identifiers in the Gregorian (western) Calendar

implicit	**two-letter**	**meaning**	**starts with**	**digits**
Y	CY	year	0	4
M	MY	month of the year	1 (January)	2
	CM	month (continuous)	0	
W	CW	week (continuous)	0	
	WY	week of the year	1	2
D	DM	day of the month	1	2
	CD	day (continuous)	0	
	DY	day of the year	1	3
J	DW	day of the week	1 (Monday)	1
H	HD	hour of the day	0	2
	CH	hour (continuous)	0	
N	NH	minute of the hour	0	2
	CN	minute (continuous)	0	
S	SN	second of the minute	0	2 with '.' and decimales
	CS	second (continuous)	0	

Examples

Paradigmatic examples	
M09	September

MY09	September (using explicit ordinal two letter
M0915	September 15
M091516	September 15 at 4 PM
M09151630	September 15 at 4:30 PM
M0915163044.12	September 15 at 4:30:34.12 PM
M01,03,07	January, March, and July
M/2	every even month
M/2%1	every odd month
M04-09	April to September
M04-09/2	every second month from april to september
J6	Saturday
J1,3,4	Monday, Tuesday, Thursday
J/2	Tuesday, Thursday, Saturday
J/2%1	Monday, Wednesday, Friday, Sunday
J1-5	Monday to Friday
J1-5/2%1	Monday, Wednesday, Friday
W/2	every other week
W/2 J6	every other Saturday
WY20	the 20th week of the year
WM2	the second week of the month
DY128	the 128th day of the year
WM2 J6	Saturday of the 2nd week of the month
M05 WM2 J6	Saturday of the 2nd week of May month
M05 DM8-14 J6	Mother’s day
Examples from above	
W/2 J2	every other Tuesday
J2	every Tuesday
J1 H0000	Monday 0:00
J1 H0900	Monday 9:00
J4 H1200	Thursday 12:00 noon
J1 H0900-1500	Mondays 9:00 to 15:00
D050000	5th day of the month at 0:00

D05	entire 5th day of the month from 0:00 to
H0800-0900	time of day 8:00 to 9:00 is given as
M02	entire month of February
M0415 --> Minneapolis M0915 --> Arizona	The Snowbird’s address that is different between April 15 and September 15 than over the Winter
H0930-1945 --> office H2030-0900 --> home	telephone numbers from 9:30 to 19:45 (office) and from 20:30 to 9:00 (home)
J1-3 H0800-1600 + J4 H0800-2100 + J5 H0800-1200	business hours shown in the introductory example
W/2 J6 H0830-1200	every other Saturday from 8:30 to 14:00
M04150000-10142400	Snowbird who is out of Arizona between April 15 0:00 and October 14 24:00 exact to the minute

The literal expression provides a short human-comprehensible notation for periodic time points and intervals. A computer can translate the literal notation into the period/phase notation relatively easily. The Snowbird example is much simpler and more precise to state in the literal form than in the period/phase form. Mother’s day, however, still comes with the same difficulty. Note that the second Saturday of the month is not the same as Saturday of the second ordinal week of the month (e.g. if the month starts on a Sunday, the second saturday is in the third ordinal week of the month.)

Besides defining periodic cycles, a calendar defines irregular events, such as holidays. For office hours, one sometimes need to refer to holidays. Even though, there are rules to determine holidays for a calendar, some involving even more cycles (e.g., Easter and the phases of the moon,) we can not expose these rules in our time expressions. Rather we need to have a shorthand way to refer to holidays. We do so using the period identifiers starting with ’J’. The following codes are defined for the Gregorian calendar:

Period Identifiers for Holidays

JH	holiday
JH’EAS’	the Easter holiday
JB	regular business day (Monday to Friday excluding holidays)
JE	regular weekend (Saturday and Sunday excluding holidays)

For example, opening hours may be every second Saturday from 8:30 to 14:00 if that Saturday does not fall on a holiday: "`JE J6 H0830-1400`".

Holidays can be named using a code in single quotes. Such holiday codes are highly localized and should be defined locally. Holiday rules can involve a second non-business day the Friday before or the Monday after a holiday that falls on a weekend. Those holidays should also be coded in order to avoid such complexity of holiday rules to be exposed in the literal expression. For the U.S. the following holiday table is defined:

`XME`	Christmas Eve
`XMS`	Christmas
`NEW`	New Year
`GFR`	Good Friday
`EAS`	Easter
`PEN`	Pentecoste
`PRE`	Presidential Day
`MEM`	Memorial Day
`MEM5`	Friday before Memorial Day Weekend
`MEM1`	Monday after Memorial Day Weekend
`JL4`	4th of July
`JL45`	Friday before 4th of July Weekend
`JL41`	Monday after 4th of July Weekend
`LBR`	Labor Day
`LBR1`	Friday before Labor Day Weekend
`LBR5`	Monday after Labor Day Weekend

Other Friday-before and Monday-after codes may need to be added, but can be constructed in the stereotypical way by appending a 5 for Friday and a 1 for Monday. Other countries, and, as in Germany, different states or provinces will have other holidays. Most western countries share the major christian holidays, which need not be redefined.

4.6.5 Other Issues and Curiosities About Time

"I got sick at my birthday, about 20 years ago," is an expression that we might want to capture. One possible representation for this time would be "yyyy0219" if my birthday is February 19th and if yyyy is constrained to *this year* - yyyy is approximately 20 years. If from another source we gather that I got sick in "1976", but don’t know the exact month and day, then we can conclude that I got sick in "19760219", because 1998 - 1976 = 22. This seems a somewhat rare use case, but definitely worth considering.

"I got that cough in spring," might lead us to adjust probabilities for pollen allergy. The season of the year is of interest in epidemiology. Bob Dolin, in his JAMIA Article on *Modeling the temporal complexities of symptoms*, suggests accounting for "season" in time expressions. The difficulty here is that seasons depend on the geographical latitude and we can not infer the season from the month of the year. January is Summer in Australia, South Africa, Chile, and Argentinia while northern folks assume that January is the worst part of the Winter. Moreover, at the equator there are not the usual four seasons, however, in tropical regions, there is the Monsun season, which may be considered one of two seasons, or a fifth season. Rather than refering to the season symbolically one should attempt to capture an uncertain date with a variance of about two months.

Noteworthy references on time expressions are CEN TC251's ENV 12381 *Health care informatics; time standards for health care specific problems* and the ARDEN Syntax. Those two standards not only define relations and operators on time values but also on events and episodes which are related in time.

Relative times of the semantics NOW + duration offset stick out as the most prominent feature defined by those and other time related standards. We might thus consider the ability to specify relative time. Some conventions use expressions like "t-1" to mean "yesterday". Relative time expressions are of the data type point in type, but the exact value depends on a parameter (the actual time) specified elsewhere. However, the use case of relative time in data communication and data storage seems unclear, since one needs to fixate NOW at some point.

5 Orthogonal Issues

There are variations of meaning that can apply to many different data types. Such variations are forming ranges, adding comments, specifying a validity period or a history of some data element, and, of course, specifying uncertainty about some information. Rather than define specific ways for every data type to express such semantic variations, this type system uses generic types [p. 12] combined with implicit type conversion [p. 18] to yield a similar effect as was used in HL7 2.x to modify existing data types.

HL7 2.x used to append new optional components at the end that served as modifiers of the meaning of the prior components. Thus the same message element instance could conform to more than one type, the base type and the extended type.

In a strong type system we can yield the same effect through generic types [p. 12] combined with implicit type conversion [p. 18] . This method virtually "overlays" extended types on top of the base types.

5.1 Interval

Interval (IVL)			
Generic data type to express a range of values of the base data type. An interval is a set of consecutive values of any totally ordered data type. An interval is thus a continuous subset of its base data type.			
GENERIC TYPE			
parameter name	**allowed types**	**description**	
T	*OrderedType*	Any ordered type can be the basis of an interval. It does not matter whether the base type is discrete or continuous, or whether any algebraic operators are defined for that type.	
component name	**type/domain**	**optionality**	**description**
low	T	**optional**	The lower boundary.
low closed	Boolean	**required** default *false*	Indicates whether the interval is closed or open at the lower boundary. For a boundary to be closed, a finite boundary must be provided, i.e. unspecified or infinite boundaries are always open.
high	T	**optional**	The upper boundary.
high closed	Boolean	**required** default *false*	Indicates whether the interval is closed or open at the high boundary. For a boundary to be closed, a finite boundary must be provided, i.e. unspecified or infinite boundaries are always open.
width	dif(T)	**required** mostly derived	For base types with a difference operation, the width is the difference between high and low boundary. When both boundaries are known, width is a derived value. When one bounday and the width is known, the other boundary is also known. When no boundary is known, the width may still be known. For example, one knows that an activity takes about 30 minutes, but one does not need to know when that activity is carried out. For a pure ordinal base type without a difference operation, the width is the cardinality of the interval.

Ranges or intervals of values are most abundant as ranges of absolute time, used for ordering and scheduling. Note that an interval is **not** to be used to specify confidence intervals for uncertain values.

We use the terms "range" and "interval" interchangably as synonyms. Webster’s dictionary defines:

range

> 1 a (1) : a series of things in a line, [...]

interval

> 1 a : a space of time between events or states [...]
> 3 : a set of real numbers between two numbers either including or excluding one or both of them

Thus, in common language interval and range are not quite synonyms. A range is the ordered "line of things" while the common notion of an interval is the gap between two things. However, "interval" is used in mathematics for things being aligned in a set.

People normally use ranges for three different purposes that can be intuitively described as

1. a set of values, where each value may apply under some circumstances (e.g. an order scheduled to begin at 3:15 and end at 4 o’clock);
2. one single unknown value supposed to lie within the range of values given (e.g. a measurement which turns out to be off the lower absolute limit and therefore can be reported only as a range with an upper boundary);
3. one single value whose set of possible values is partitioned into equivalence classes because the exact differences are not interesting or not measurable (e.g in microbiologic susceptibility testing, we may have a parameter "OXACILLIN SUSC" where only the following equivalence classes are of interest: > 8.0 µg/ml (not susceptible); 4.0±2 µg/ml (limited susceptibility); and < 2.0 µg/ml (susceptible)).

The interval data type shall be primarily used when the entire set of values is meant, not just one value from that set. Notably if the motivation for considering an interval is that there is uncertainty, then the interval is the wrong choice. For uncertainty or inaccuracy one of the data types for uncertainty [p. 174] must be used instead. Thus in the above list, only item 1 is definitely a use case for intervals.

Intervals can be open or closed at either side:

[*n*, *m*] is a **closed interval**. A value *x* is an element of the interval if and only if *x* is greater **or equal** than *n* **and** less **or equal** than *m*. That is, the boundaries are included in the interval.

]*n*, *m*[is an **open interval**. A value *x* is an element of the interval if and only if *x* is greater than *n* **and** less than *m*. That is, **the boundaries *n* and *m* are not included in the interval.**

Obviously an interval can be closed on one side while open on the other side.

Intervals can have finite or infinite boundaries on either side, if the base type contains the notion of inifinity. Note that an interval with two infinite boundaries is equivalent to the entire range of an infinit base type.

One boundary of an interval may be unknown. For example, the expression "< 3" is an interval with an unknown lower boundary and an open finite upper boundary. An interval must not have both boundaries unknown.

An interval can only be closed at a finite boundary. That is, if a boundary is an *infinity* or unknown, the interval can not be closed at that boundary.

In order to treat incomplete information uniformly we must accomodate the case where only the width of an interval is known whereas both boundaries are unknown. Otherwise we would force one case of incomplete information to be represented by a different data type, and thus a different dependent attribute, which would force the constraints of dependency between the interval and its width to be handled outside. This would violate the rule of encapsulation.

The fact that the width is kept as a component of the interval illustrates once more that data type components in this specification are *semantic* components and **not** components of any particular representation. This means that if a representation of an interval is based on low and high boundary, the width will only be made explicit in the exceptional case where both (!) boundaries are undefined. Another representation may be based on low boundary and width, in which case the high boundary will only be sent in the exceptional case where low boundary and width are undefined. Every representation will have to deal with one such exceptional case though.

Although, we do distinguish between surface form and semantic components with intervals as with any other data type, we specify a character string literal form for interval expressions that is tuned toward intuitiveness and is recommended for use in character based encoding rules. Here is a mapping between surface forms (string literals) and the uniform interval form:

literal	interval form	instance notation
<= *n*	]**unk**; *n*] (low open)	`(Interval :high` *n* `:highClosed #true)`
>= *n*	[*n*; **unk**[(high open)	`(Interval :low` *n* `:lowClosed #true)`
< *n*	]**unk**; *n*[(low open and high open)	`(Interval :high` *n*`)`
> *n*	]*n*; **unk**[(low open and high open)	`(Interval :low` *n*`)`
= *n*	[*n*; *n*]	`(Interval :low` *n* `:lowClosed #true :high` *n* `:highClosed #true)`
n - *m*	[*n*; *m*]	`(Interval :low` *n* `:lowClosed #true :high` *m* `:highClosed #true)`
n -< *m*	[*n*; *m*[(high open)	`(Interval :low` *n* `:lowClosed #true :high` *m* `:highClosed #false)`
n >- *m*	]*n*; *m*] (low open)	`(Interval :low` *n* `:lowClosed #false :high` *m* `:highClosed #true)`
[*w*]		`(Interval :width` *w*`)`

Note that the column headed "interval form" does not define literals. Note also that literal forms of multiple different data types are not designed to be intermixed in a single expression. If they are, the literals need to be tagged by the data type.

As always, various constraints can be made on data types. I.e., the components of the interval data structure can be constrained to certain allowable values or combinations of values only. As a notable special case, one could constrain intervals such that any allowable value would have to have an unknown (or infinite) boundary at one side.

5.1.1 Intervals as Sets - The Notion of Set Revisited

Intervals are continuous sets of elements of the base data type. Thus intervals have a relationship with set-collections [p. 14] . Discrete intervals can be converted into an enumerated set-collection. We thus have to revisit our notion of set as defined initially. A set is no longer just an enumerated collection of discrete unordered elements. The various kinds of sets are described by the following taxonomy:

- set-collection (finite, discrete, enumerated set)
- interval (continuous ordered subset)
  - finite countable interval (e.g., integers 1-3)
  - unbounded infinite countable interval (e.g., all integers)
  - partially bounded infinite countable interval (e.g., integers > 3)
  - totally bounded infinite uncountable interval (e.g., real 0.0 - 1.0)
- periodic point in time (sparse, infinite, discrete, ordered subset of point in time)
- periodic interval of time (sparse, infinite, partially continuous, ordered subset of point in time) alternatively: set of interval of point in time.
- set derived from other sets through set operations (union, intersection.)

At this point all of the above mentioned kinds of sets are defined, except for the general derivative set that is specified as a set algebra term from other sets.

5.2 General Annotations

HL7 v2.x made abundant use of the NTE segment for notes and comments. Up until now, there is no such construct for HL7 version 3. The NTE segment was a very useful construct to communicate information that can not be communicated otherwise. NTE segments usually contain display data, meant to be shown to human users. Th v2 NTE segments had the disadvantage that they would occur only at certain places in the message. A comment in an NTE segement was scoped to parts of the message structure, however, the scope could not be narrowed down to the level of a single data element or component.

The following generic type for annotations can be overlayed over a value of any other data type. An implicit conversion rule exists that will convert any annotated *T* to a *T* at the receiver side.

Annotated Information (ANT)			
Generic data to give allow arbitrary display data annotations for any message element instance. **An annotation can not change the meaning of the annotated value and must not be used when the value would be wrong without the annotation**.			
GENERIC TYPE			
parameter name	**allowed types**	**description**	
T	*ANY*	Any message element type can be annotated.	
component name	**type/domain**	**optionality**	**description**
value	T	**required**	The information itself.
note	Display Data [p. 46]	**required**	The annotation as display daya to be eventually displayed to a user or administrator.

Note that this annotated information data type, as a Message Element Type (MET) could be used to annotate any Message Element Instance (MEI), regardless whether that MEI was derived from a RIM class, a RIM attribute, or from any component of a data type. Thus this annotated information generic type is enough to carry the NTE feature of version 2 over to version 3.

Annotations are primarily used to eventually display the annotation to human users. For instance, a lab value might be sent annotated, in which case the medical record user interface program might shows a little marker in the respective cell of the flowsheet. When the user clicks on that mark, a text box pops up that displays the display data annotation.

However, annotations in version 2 NTEs were sometimes used like a codes. This happens for three different reasons

1. instead of fixed canned notes and comments, only a single symbol is sent, as an abbreviation for the whole commen;
2. people want to save bandwidth by "compressing" longer comments into abbreviations; or
3. the notes and comments are meant to be interpreted by computers instead of humans.

To use free abbreviations or codes in NTE segments is a problematic habit, though. First of all, it is hardly interoperable, becasue one will hadly find any standard for notes and comments codes. Indeed if there were any such standard, then the use case of those codes would be so well established that it would warrant better means than just annotations. Such codes usually translate into No Information flavors, or attributess of specific classes.

There are clearly some "use cases" we deliberately will not support. There is no need for ad-hoc "compression" of data using such abbreviations. The problems and overhead that such abbreviations put on the message processing side outweighs by far the minor saving of bandwidth. Also, we do not want to support the use case of lazy message creation. Indeed many coded annotations fall in the category of lazy message creation, where data could be sent in appropriate message fields.

If there are use cases for coded annotations that are not supported by the RIM or the data type model, those should be fed back into the HL7 development process. Codes that people used to send in NTEs should be systematized and used to improve the HL7 version 3 data models and messages.

It might have been reasonable in v2.x to use those coded NTE segments for this purpose, in v3 we definitely want to use the available stanardized information structure. If any significant amout of real existing annotation could not be accomodated in RIM data elements, we should drive a use case analysis from there suggesting improvements to the RIM.

5.3 The Historical Dimension

In the recent years HL7 has experienced a need for data elements to be communicated with a history. I.e. the National Immunization Program (CDC, State Departments of Health) needed to communicate historic address information. Other examples for history are "effective date" and "end date" of identifiers or other data. The traditional approach to this problem was to extend a preexisiting data type *T* or to create a new data type X-*T*. Using generic types as described above, we no longer need to take care of history information for every existing type. Instead we can define the following set of generic types:

5.3.1 Generic Data Type for Information History

History (HIST)		
Generic data type to give the history of some information. This is an ordered list of data of the same type along with the time interval giving the time the information was (or is) valid. The order of history items in the lists should be backwards in time. The history information is not limited to the past history, expected future values can also appear.		
GENERIC TYPE		
parameter name	**allowed types**	**description**
T	*ANY*	Any data type can be used here.
ORDERED LIST OF History Item [p. 174] <T>		

5.3.2 Generic Data Type "History Item"

History Item (HXIT)			
Generic data to give the time range in which some information was, is, or is expected to be valid.			
GENERIC TYPE			
parameter name	**allowed types**	**description**	
T	*ANY*	Any data type can be used here.	
component name	**type/domain**	**optionality**	**description**
value	T	**required**	The information itself.
validity period	General Set [p. 170] <Point in Time [p. 148] >	**required**	The set of time the given information was, is, or is expected to be valid. This set of time can be a simple interval of time or a periodic point or interval of time for cyclic events. The interval can be open or closed infinite or undefined on either side.

5.4 Uncertainty of Information

Uncertainty may exist for all kinds of informations. Information is selection of a signal (value) from a set of possible signals (values). Uncertain information is selection of several values from a set of possible values where we assign to every value a probability (i.e. belief that the given information applies). We may distinguish four cases:

1. There are only two possible values where one is the negation of the other (boolean). In that case we need to specify a probability *p* for only one value (preferably the value meaning "true"). The probability of the other value is then 1 - *p*.
2. The set of possible values may have no total order. In that case we have to send pairs of <*value*, *probability*>.
3. The set of possible values may have a total order but is discrete. In that case, we can send <*value*, *probability*> pairs too. In addition, however, there is a mapping of the set to the set of natural numbers, and we can specify a discrete probability distribution (e.g., binominal, geometric, poisson) and the necessary parameters of those distributions.
4. The set of possible values may have a total order but is continuous. In that case, we **can not** send <*value*, *probability*> pairs. But we can select a continuous probability distribution (e.g., normal, uniform, gamma, chi-square) and its necessary parameters.

The following are examples of where uncertainty appears in the language of medical practice

- A pathologist says: "There is a 30% probability that this lesion is malignant."
- A pathologist says: "This lesion is malignant." A medical record system may find out from case-based reasoning (experience) that if pathologist A discovered malignancy, he was right in 80% of the cases, whereas if pathologist B makes the same statement, he was right in only 70% of the cases.
- A pathologist says: "This lesion is probably malignant." Again from experience, a system can say that if the word "probably" was used the chance of malignancy is 40% (whereas if this pathologist had said "could be" the chance would have been only 10%).

One might concluded that one needs to distinguish whether a probability was issued by the "user" or a such a system that keeps track of experiences with the pathologist’s judgment.

One might further concluded that an expression of an uncertain discrete value (e.g., malignancy) should include both, a coded qualifier of confidence and a numeric probability, where each may be assessed by different entities.

The seemingly important distinction between "user assessed" probability and "system assessed" probability suggests that every uncertain information item may be associated with many uncertainty qualifiers, each in the eye of another entity. Indeed soem piece of information may be believed at a different level of confidence by different people. Bayesian probabilities are subjective, and thus, any probability is valid only in the context of the one who issued the probability.

Uncertainty assessments (probabilities) are subjective. Thus they depend on *who* states them. For example, if I am 70% sure that what I see in the microscope are malignant cells, I express my views as such. If some experienced pathologist says that probability for malignancy is 70%, she expresses her view as such. Any receiver of that information must draw his own conclusions based on his trust in my or the pathologist’s judgment.

Practically, a receiver might apply a penalty of 0.5 to what I say, whereas the pathologist’s views would be trusted at a level of 0.95. Thus from my statement, the receiver may infer a probability of 35% for malignancy while the pathologists statement may be transformed to 67%. If the receiver has both of our statements, he may want to apply a noisy-or and infer his probability as 1-(1-35%)(1-67%) = 79%.

The bottom line is: the newly created value-probability-pair would be part of a new observation assessed by the receiver of both mine and the pathologists statements, penalized and combined by the receiver. The receiver drafted his judgment about the case from information received by others, but he has drawn his own conclusions and is responsible for them. This shows that there is not one correct proability that would "objectively" qualify any given statement.

When this newly drafted value-probability-pair is communicated further along to someone else, the sender may or may not quote both of his input-statements plus his own conclusion. In any case, the receiver of that information would again penalize and combine what he has got based on his trust in the judgment of the originators of the incoming statements.

It generally doesn’t matter whether a probability was issued by a human "user" or by any kind of decision support "system". The same rules apply: the probability is subjective and the receiver has a responsibility to value the uncertain information he received. Knowing the originator of the uncertain statement is essential (as it is always essential to know *who* said what), but knowing just the category "user" vs. "system" does not help.

A data type for uncertain information should, however, not include implied associations between RIM classes to suit the need for attributions of probabilities. Thus, one uncertain value should not be attributed to some Healthcare_provider instance of the RIM. For example, we should not build a data type composed of the triple <*value*, *probability*, *originator*>, where *originator* would be a foreign key to some Stakeholder or Healthcare_provider. Rather, the uncertain value would be included in a RIM class instance, where the attribution or responsibility of the statement is clear from that context of the RIM class.

It is true that any instance of uncertain information must be attributed to an originating entity (Doctor or decision support system) just like a "supposedly certain" information must be attributed. But attribution of information is outside the scope of this data type model, since attribution is modeled properly in the RIM.

New issues that the editor believes are much more important

A much more important open issue is the relationship between sets, bags, intervals and periodic sets and uncertainty. It appears as if general notion of a set can be used where multiple possible values exist without any particular probability distribution. This would translate to the uniform probability distribution over the set. The question is whether the data type definitions for probability distributions should not be better aligned to the notion of sets.

An second related issue is the fact that we sometimes want to use a probability distribution (parametric or non-parametric) in order to describe a frequency distribution. Sometimes laboratory observations on population samples are reported in such a "consolidated" way using histograms. Although the distinction between "probability" and "frequency" is blur, the wording in this specification may need to be changed to invite the probability constructs to be used for frequencies as well.

A third related issue is whether we want to support other "weights" of certainty and importance that have become well-known in the decision support community. Examples are the weights of logistic regression and neural nets, all kinds of plausibility measures (Dempster-Shafer possibilities, Fuzzy membership functions, Shortliffe’s certainty factors, etc.), and the heuristic numbers used in Internist-I/QMR (evoking strength, frequency, import), or Medcin and others.

A third related issue is that the probability distributions and especially the parametric probability distribution can be used to describe distribution quantities other than probabilities. For example, a probability distribution "multiplied with" a flow rate may describe the setting of a ventilator. Should we extend our definition to embrace quantities that are neither probabilities nor frequencies nor any other uncertainty measure?

5.4.1 Uncertain Discrete Values

Discrete values can be assigned a single probability number to express the level of confidence that the given information applies

Uncertain Discrete Value using Probabilities (UDVP).			
Generic data type to specify one uncertain value as a pair of <*value, probability*>.			
GENERIC TYPE			
parameter name	**allowed types**	**description**	
T	*DiscreteType*	Any data type that is discrete can be used.	
component name	**type/domain**	**optionality**	**description**
value	T	**required**	The value to which a probability is assigned.
probability	Real Number [p. 136] 0.0 to 1.0.	**required**	The probability assigned to the value.

Many people are reluctant to use probabilities to express their subjective belief, because they think that such a probability is not "exact" enough, or that a probability would have to be derived somehow experimentally. While this is true in the "frequentist" sense, frequentist probabilities never hold for individual cases, only at average in a population.

Bayesian probabilities, on the other hand, do not have to be "exact", especially one does not need to carry out a series of experiments (samples) in order to specify a probability. Probabilities are always estimated (frequentist probabilities are estimated as well). Bayesian probability theory equals the notion of "probability" with "belief". The probability is thus an assessment of the subjective belief of the originator of a statement. Some subjective numeric probability is often better than a mere indicator that a value is "estimated".

Probabilities are always subjective. Just like any other information, uncertain information needs to be seen in the context of who gave that information (attribution). A recipient updates his knowledge about a case from the received uncertain information based on how much confidence he has in the judgment of the originator of the information.

Both elements in the value-probability-pair are part of the statement made by one specific originator. Along a chain of communication, one value may be reported by different entities and assigned a different probability by each of them.

This data type does not allow to make specific attributions to originators of the information. The rules of attribution are the same whether information is given as uncertain or certain/precise. In particular, in case information is given in an instance of a RIM Service_event class, the attribution is provided by the Stakeholder designated as the active participation of type "originator of the information". For "slotted" data elements (PAFM), implicit attribution defaults to the sending system.

5.4.2 Non-Parametric Probability Distribution

If the domain of a discrete value contains more than two elements, one might want to specify probabilities for more than one element. This can be done using a non parametric probability distribution. A non parametric probability distribution is a collections of alternative value-proability-pairs.

Non-Parametric Probability Distribution (NPPD)		
Generic data type to specify an uncertain discrete value as a set of <*value*, *probability*> pairs (uncertain discrete values [p. 177]). The values are considered alternatives and are rated with probabilities for each of the values to apply. Those values that are in the set of possible alternative values but not mentioned in the non-parametric probability distribution data structure will have the rest probability distributed equally over all unmentioned values. That way the base data type can even be infinite (with the unmentioned values being just neglected).		
GENERIC TYPE		
parameter name	**allowed types**	**description**
T	*Discrete*	Any data type that is discrete can be used. Usually we would use non-parametric probability distributions for unordered types only and only if we assign probabilities to a "small" set of possible values. For other cases one may prefer parametric probability distributions.
SET OF Uncertain Discrete Value using Probabilities [p. 177] <T>		

The values in a discrete probability distribution are generally considered alternatives. It is understood that only one of the possible alternative values may truly apply. Because we may not know which value it is, we may state probabilities for multiple values. This does not mean that the values would in some way be "mixed." However, when Rough Sets theory or Fuzzy Logic is used as the underlying theory of uncertainty, the difference between "alternative" and "mixed" values becomes blur. Friedman and Halpern (1995) have shown that all of those theories for uncertainty (probability, rough sets, fuzzy logic, Dempster-Shafer) can be subsumed under a theory of "plausibility". This theory of plausibility would of course be open as to whether or not a distribution is considered over alternative values as opposed to a mixture of the values.

However, probability is the most widely understood and deployed theory (although fuzzy logic decision support systems are used in clinical medicine). If some value should be represented as a "mixture" of a set of categorial values, other means should be investigated before resorting to "plausibility" theory. For instance, suppose we have to decide about a color in the code system "red, orange, yellow, green, blue, purple". Probabilistically all those values would be alternatives and thus a given color may be stated "orange with a probability of 60%", but the alternatives red and yellow are also considered with probabilities 20% and 15% resp. More naturally we would like to "mix" the colors saying that the color we see is 60% orange, 20% red, 15% yellow and 5% green. We could use fuzzy logic to do that, but a more easy to understand approach would be to use a more appropriate color model than the list of discrete codes. A more appropriate color model would, for instance, be the RGB system, where every color is represented as a mixture of the three base colors red, green and blue (or magenta, yellow, and cyan in subtractive color-mixing).

Type cast rules allow conversion between a singular uncertain discrete value using probabilities and non-parametric probability distribution and vice versa.

A bag-collection can be cast to a non-parametric probability distribution, where the probabilities for each item of the bag are the quotient of the count of that item devided by the size of the bag.

An example for a discrete probabilities would be a differential diagnosis as a result of a decision support system. For instance, for a patient with chest discomfort, it might find the following probability distribution:

```
(NonParametricProbabilityDistribution
  (SET :of UDV-P
     (UDV-P
        :value "myocardial infarction"
        :probability 0.4)
     (UDV-P
        :value "intercostal pain, unsp."
        :probability 0.3)
```

```
       (UDV-P
          :value "ulcus ventriculi sive duodeni"
          :probability 0.1)
       (UDV-P
          :value "pleuritis sicca"
          :probability 0.1)))
```

This is a very compact representation of information that could (and should in general) be communicated separately using Clinical_observation or Health_issue class instances (or OBX-segments in v2.3). However, there are advantages of using the data type for non-parametric probability distribution:

- it is much more compact;
- it is immediately clear that the stated values are alternatives assessed by one originator of the observation;
- it is clearly specified from the definition of the data type that there is a rest-probability of 0.1% that is not assigned to any of the other diagnoses.

Those facts would be hard to discover from a bunch of Health_issue class instances.

The Health_issue class instances could in some way be linked together to express the same distribution. This would be the method of of choice if one wishes to track down more precisely how the alternative differential diagnoses have been confirmed or otherwise clinically addressed. For the purpose of patient care the expanded set of Health_issue instances would be clearly more useful. However, as an excerpt summary of a decision support process, the short form is useful too.

5.4.3 Parametric Probability Distribution

For continuous values it is not possible to assign a probability to every single value. One can assign a probability to an interval of consecutive values (confidence inteval), however, the confidence interval can be calculated from a continuous probability distribution.

The data type for continuous probability distributions allows to choose from a large menu of distribution types commonly used in statistics. Every distribution type has specific paramters. However, for compatibility with systems that do not understand a particular distribution type, the mean and standard deviation must always be given.

Parametric Probability Distribution (PPD)			
Generic data type to specify an uncertain value of an ordered data type using a parametric method. That is, a distribution function and its parameters are specified. Aside from the specific parameters of the distribution a mean and standard deviation is always specified to help maintain interoperability is receiving applications can not deal with a certain the probability distribution. The base data type may be discrete or continuous. Discrete ordered types are mapped to natural numbers by setting their "smallest" possible value to 1, the second to 2, and so on. The order of non-numeric types must be **unambiguously** defined.			
GENERIC TYPE			
parameter name	**allowed types**	**description**	
T	*OrderedType*	Any ordered type (anything that is unambiguously mapped to numbers) can be the basis of an uncertain quantity. Examples are Integer Number [p. 135] , Real Number [p. 136] , and PhysicalQuantity [p. 141] .	
component name	**type/domain**	**optionality**	**description**
mean	T	**required**	The mean (*expected value* or *first moment*) of the probability distribution. The mean is used to standardize the data for computing the distribution. The mean is also what a receiver is most interested in. Applications that can not deal with distributions can still get the idea about the described quantity by looking at its mean.
standard deviation	dif(T)	**required**	The standard deviation (*square-root of variance* or *square-root of second moment*) of the probability distribution. The standard deviation is used to standardize the data for computing the distribution. Applications that can not deal with distributions can still get the idea about the confidence level by looking at the standard deviation.
type	Code Value [p. 58]	**required**	The type of probability distribution. Possible values are as shown in the attached table.
parameters ...			

The number of parameters, their names and types depend on the selected distribution and described in the attached table. This table will define component names to be used in the above data type definition.

Distribution types, their mean and parameters.

type	**description and parameters**			
	symbol	**name or meaning**	**type**	**constraint or comment**
guess	Used to indicate that the mean is just a guess without any closer specification of its probability. This pseudo distribution does not have any parameter aside from the expected value and standard deviation.			
	E	mean		
	V	variance		
DISTRIBUTIONS OF DISCRETE RANDOM VARIABLES				
binominal	Used for *n* identical trials with each outcomes being one of two possible values (called success or failure) with constant probability *p* of success. The described random variable is the number of successes observed during *n* trials.			
	n	number of trials	Integer	$n > 1$
	p	probability of success	Real	*p* between 0 and 1
	E	mean		$E = n p$
	V	variance		$V = n p(1 - p)$
geometric	Used for identical trials with each outcomes being one of two possible values (called success or failure) with constant probability *p* of success. The described random variable is the number of trials until the first success is observed.			
	p	probability of success	Real	*p* between 0 and 1
	E	mean		$E = 1 / p$
	V	variance		$V = (1 - p) / p^2$

negative binominal	Used for identical trials with each outcomes being one of two possible values (called success or failure) with constant probability *p* of success. The described random variable is the number of trials needed until the *r*th success occurs.			
	p	probability of success	Real	*p* between 0 and 1
	r	number of successes	Integer	$r > 2$
	E	mean		$E = r / p$
	V	variance		$V = n\ r\ (N - r)\ (N - n) / (\ N^3 - N^2\ )$
hypergeometric	Used for a set of *N* items, where *r* items share a certain property P. The described random variable is the number of items with property P in a random sample of *n* items.			
	N	the total number of items	Integer	$N > 1$
	r	number of items with property P	Integer	$r > 1$
	n	sample size	Integer	$n > 1$
	E	mean		$E = (n\ r) / N$
	V	variance		$V = r(1 - p) / p^2$
Poisson	Describes the number of events observed in one unit that occur at an average of *lambda* per unit. For example, the number of incidents of a certain disease observed in a period of time given the average incidence of *E*. The Poisson distribution only has one parameter, which is the mean. The standard distribution is the square-root of the mean.			
	E	mean		
	V	variance		$V = E$
DISTRIBUTIONS OF CONTINUOUS RANDOM VARIABLES				

uniform	The uniform distribution assigns a constant probability density over a range of possible outcomes. No parameters besides mean E and standard deviation s are required. Width of the interval is sqrt(12 V) = 2 sqrt(3) s. Thus, the uniform distribution assigns probability densities $f(x) > 0$ for values E - sqrt(3) s >= x <= E + sqrt(3) s and $f(x) = 0$ otherwise.			
	E	mean		$E = (\mathit{low} + \mathit{high}) / 2$
	V	variance		$V = (\mathit{high} - \mathit{low})^2 / 12$
normal Gaussian	The well-known bell-shaped normal distribution. Because of the central limit theorem the normal distribution is the distribution of choice for an unbounded random variable that is an outcome of a combination of many stochastic processes. Even for values bounded on a single side (i.e. greater than 0) the normal distribution may be accurate enough if the mean is "far away" from the bound of the scale measured in terms of standard deviations.			
	E	mean		often symbolized μ
	V	variance		often symbolized sigma^2
gamma	Used for data that is skewed and bounded to the right, i.e. where the maximum of the distribution curve is located near the origin. Many biological measurements, such as enzymes in blood, have a gamma distribution.			
	alpha		Real	$\mathit{alpha} > 0$
	beta		Real	$\mathit{beta} > 0$
	E	mean		$E = \mathit{alpha}\ \mathit{beta}$
	V	variance		$V = \mathit{alpha}\ \mathit{beta}^2$
chi-square	Used to describe the sum of squares of random variables which occurs when a variance (second moment) is estimated (rather than presumed) from the sample. The chi-square distribution is a special type of gamma distribution with parameter $\mathit{beta} = 2$ and $\mathit{alpha} = E / \mathit{beta}$. The only parameter of the chi-square distribution is thus the mean and must be a natural number, so called the *number of degrees of freedom* (which is the number of independent parts in the sum).			
	n	number of degrees of freedom	Integer	$n > 0$
	E	mean		$E = n$
	V	variance		$V = 2n$

Student-*t*	Used to describe the quotient of a standard normal random variable and the square-root of a chi-square random variable. The *t*-distribution has one parameter *n* which is the number of degrees of freedom.			
	n	number of degrees of freedom	Integer	$n > 0$
	E	mean		$E = 0$ (the mean of a standard normal random variable is always 0)
	V	variance		$V = n \ / \ (\ n - 2 \ )$
F	Used to describe the quotient of two chi-square random variables. The *F*-distribution has two parameters n_1 and n_2 which are the numbers of degrees of freedom of the numerator and denominator variable respectively.			
	n	numerator's number of degrees of freedom	Integer	$m > 0$
	m	denominator's number of degrees of freedom	Integer	$m > 0$
	E	mean		$E = m \ / \ (\ m - 2 \ )$
	V	variance		$V = 2m^2 \ (m + n - 2) \ / \ (\ n(m - 2)^2 (m - 4) \ )$
logarithmic normal	The logarithmic normal (log-normal) distribution is often used to transform skewed random variable X into a normal form $U = \ln X$. The log-normal distribution has the same parameters as the normal distribution.			
	μ	mean of the resulting normal distribution	Real	
	sigma	standard deviation	Real	
	E	mean of the original skewed distribution		$E = e^{\ \mu + 0.5 \ sigma^2}$
	V	variance of the original skewed distribution		$V = e^{\ 2\mu + sigma^2} \ (\ e^{sigma^2} - 1 \ )$

beta	The beta distribution is used for data that is bounded on both sides and may or may not be skewed. Two parameters are available to adjust the curve.			
	alpha		Real	$alpha > 0$
	beta		Real	$beta > 0$
	E	mean	T	$E = alpha / (alpha + beta)$
	V	variance	T	$V = alpha\ beta / ((alpha + beta)^2 (alpha + beta + 1))$

The distribution type "guess" can be used in two different ways

1. a value is known to be uncertain but no information exists about the dispersion of the probability distribution. In this case, no standard deviation is provided.

2. a value is known to be uncertain and a dispersion is approximately known, but no information exists about the distribution type. For example, the common expression "Age: 75±10 years" would be mapped to a distribution type of guess with standard deviation set to 5 years. This seems to pretend a normal distribution, but it does not. Using 10/2 as the standard deviation is just a convention.

The *mean* component is mentioned explicitly. This component will be used in type casting a probability distribution over type T to a simple value of type T in a case where a receiving application can not deal with or is not interested in probability distributions.

The literature on statistics commonly lists the mean as dependent on the parameters of the probability distributions (e.g. the mean of a binominal distribution with parameters *n* and *p* is *np*. Because we choose to mention the mean (to help in roughly grasping the "value") the parameters of the distributions may be defined in terms of the mean.

In the above table, the dependencies between the explicit components mean and standard deviation and the parameters of the distribution are not always resolved. If we want to give mean and standard deviation explicitly there will often be redundancy in the parameters. However, it seems to be useful to let people specify parameters in the natural way rather than dependent on mean and standard deviation. [needs revision]

For example, in the table above, the uniform distribution was specified based on the mean and standard deviation component without further parameters. This does **not** mean that the standard deviation component contains the half-width of the uniform distribution.

If there is redundancy in the parameters, it is an error if the specified mean and standard deviation contradict what can also be derived from the distribution and its parameters.

The type dif(T) is the data type of the difference of two values of type T. Often, T is the same as dif(T). For the data type T = Point in time, dif(T) is not Point in time but a Physical Quantity in the dimension of time (i.e. units seconds, hour, minutes, etc.). This concept is generalizable since it describes the relationship between corresponding measurements on ratio-scales vs. interval-scales (e.g., absolute (Kelvin) temperatures vs. Celsius temperatures).

Most distributions are given in a form where only natural numbers or real numbers are acceptable. If distributions of measurements (with units) are to be specified, we need a way to remove the units for the purpose of generating the distribution and then reapply the units. For instance, if $Q = \mu \mathbf{u}$ is a measured quantity with numeric magnitude μ and unit $\mathbf{u}$, then we can bind the quotient $Q / \mathbf{u}$ to the random variable and calculate the distribution. For each calculated number x_i, we regain a quantity with unit as $Q_i = x_i \mathbf{u}$.

Most distributions are given in a "standard" form, that is with mean or left boundary equals 0 and standard deviation equals 1 etc. Therefore one has to standardize the quantity to be described first. This is similar to the problem of removing and reapplying units. The method is also similar and can be unified: a transformation transforms the numeric value to a standard form and later re-transforms the standard form to the numeric value. Two issues must be considered:

- translation, i.e. moving the mean (or left boundary) into the origin (zero-point)
- scaling the value to adjust the standard deviation to one.

This means, that any transformation of a value x to a normalized value y can be described as:

$$y = (x - o) / s$$

We can combine the way we deal with the units and the standardization of the value into one formula:

$$y = (Q_i - \mu \mathbf{u}) / (s \mathbf{u})$$

Here $\mu \mathbf{u}$ is the expected value (mean) E expressed in the base type T (i.e. a Physical Quantity [p. 141]). This is further justification that we should indeed carry the mean $\mu \mathbf{u}$ and the standard deviation $s \mathbf{u}$ as an explicit components, so that scaling can be done accordingly. The product $s \mathbf{u}$ is the standard deviation (square root of the variance) of the described value. The standard deviation is a component that an application might be interested in even if it can not deal with a "chi-square" distribution function.

It would be awesome if we could define and implement an algebra for uncertain quantities. However, the little statistical understanding that I have tells me that it is a non-trivial task to know the distribution type and parameters of a sum, or product of two distributions or from the inverse of a distribution.

5.4.4 Uncertain Value using Narrative Expressions of Confidence

Uncertain Value using narrative expressions of confidence (UVN)			
Generic data type to specify one uncertain value as a pair of <*value*, *qualifier*>. The *qualifier* is a coded representation of the confidence as used in narrative utterances, such as "probably", "likely", "may be", "would be supported", "consistent with", "approximately", etc.			
GENERIC TYPE			
parameter name	**allowed types**	**description**	
T		Any data type that is allowed here, discrete or continuous.	
component name	**type/domain**	**optionality**	**description**
value	T	**required**	The value to which an uncertainty qualifier is assigned.
confidence	Concept Descriptor [p. 72]	**required**	The confidence assigned to the value.

Like it or not, we do have the use case that data is known to be just estimated and we may want to signal that the data should be relied in with caution, without having any numeric probability. This occurs most frequently when textual reports are coded.

We also have to deal with narrative expressions of uncertainty that are heard everywhere; and we may want to capture those ambiguous and largely undefined qualifiers of confidence. This is almost like an annotation to a value considered to be understood mainly by humans.

We do not specify a closed list of codes to be used. Jim Case has an action item to submit a dozen or so of qualifiers he commonly has seen, others are invited to contribute as well.

No special effort is made to assign numeric probabilities to the codes nor even to specify an order in the set of codes. Translation to numeric probabilities is not trivial, as there may be linear or logarithmic scales useful in different circumstances.

We generally discourage to use narrative expressions of uncertainty rather than numeric ones. People should be reminded over and over again that probabilities are subjective measures of belief and that an "inexact" numeric probability is much more useful than a statement that "X *is likely* to be true". Coded probabilities have no reliable meaning. Not even the order of narrative confidence qualifiers is clear in all cases (e.g., is "A is likely" more or less likely that "probably A"?) However, such coded confidence qualifiers do at least uncover the ambiguity that exists (whether we want it or not.)

Only in cases where no numeric probabilities are available (e.g. coding of narratives) should the narrative expressions of confidence be used.

Appendix A: All Data Types At a Glance

The following is an overview of the data type that we have defined so far.

Boolean (BL) [p. 24]

A boolean value is the domain of two valued logic: either true or false *tertium non datur* and all the stuff everyone should know about logics. The boolean type is amaizingly useful throughout all layers of abstraction, from the bit in a machine up to object oriented data analysis.

No Information (NULL) [p. 27]

A No Information value can occur in place of any other value to express that specific information is missing and how or why it is missing. This is like a NULL in SQL but with the ability to specify a certain flavor of missing information.

Character String (ST) [p. 38]

A character string is a primitive data type that contains Unicode characters. A single character is not considered an HL7 data type. Note that the string type is not limited to ASCII characters and none of the "escape" sequences of v2.3 are defined. Transmitting Unicode characters is considered an ITS layer issue and the application layer is not supposed to deal with the peculiarities of different character encodings.

Display Data (DD) (was: Free Text, FTX) [p. 46]

Display Data may be anything from a few formatted characters to complex documents or images. This data type is defined similar to the HL7 v2.3 ED data type that in turn is based on the MIME standard.

Technical Instance Identifier (TII) [p. 84]

Technical instance identifiers are unique and unravelable through the consistent and required use of the ISO OBJECT IDENTIFIER (OID) [p. 85] .

Technical Instance Locator (TIL) [p. 90]

A technical instance locator is a reference to some technical thing (e.g., image, document, telephone, e-mail box, etc.) It is a generalization of the well-known URL concept.

Postal and Residential Address (AD) [p. 103]

This Address data type is used to communicate postal addresses and residential addresses. The main use of such data is to allow printing mail labels (postal address), or to allow a person to physically visit that address (residential address). An address consists of tagged Address Parts [p. 104] .

Person Name (PN) [p. 113]

This type used in the RIM class Person_name that will be developed from the class Person_alternate_name of RIM 0.88 jointly with PAFM. Person names consist of tagged Person Name Parts [p. ??] . Typical name parts that exist in about every name are given names, and familiy names, other part types may be defined culturally.

Organization Name (ON) [p. ??]

A collection of organization name variants [p. ??] . Every Organization Name Variant represents an organization name used in different contexts or for a different purpose or at a different time.

Code Value (CV) [p. 58]

A code value is used to refer to technical concepts and is also the basic building block for construcing more complex concept descriptors for real world concepts.

Concept Descriptor (CD) [p. 72]

Concept descriptors are the way to refer to real world concepts (e.g. diagnoses, procedures, etc.). Just as with the old CE data type one can specify a code from one coding system with its translation into another coding system. This data type is more general than the CE so that multiple Code Translations [p. 73] can be given, and their dependencies can be exactly specified. With Code Phrases [p. 74] one single axial code can be mapped to multiple codes for a multi axial codeing system and vice versa.

Integer Number (INT) [p. 135]

Embody the usual concept of integer numbers. Integers are used almost only for counts or values derived from counts by addition and subtraction.

Real Number (REAL) [p. 136]

Embody the abstract concept of real numbers. Real numbers have a built-in notion of precision in terms of the number of significant decimal digits.

Ratio of Quantities (RTO) [p. 139]

A quotient of any two quantities. Quantities currently defined are

- Integer Number [p. 135]
- Real Number [p. 136]
- Physical Quantity [p. 141]
- Monetary Amount [p. 142]
- Point in Time [p. 148] (although those quantities are on difference scales, not ratio scales).

Physical Quantity (PQ) [p. 141]

A physical measurement with units.

Monetary Amount [p. 142]

An amount of money in a certain currency unit.

Point in Time (TS) [p. 148]

A difference scale quantity in the physical dimension of time. Usual expressions of points in time are made based on calendars, which are quite complex "coordinate systems" for time. This is basically the old "TS" data type.

Periodic Point in Time [p. 153]

A sparse set of points in time that describes periodic events, e.g., every Friday morning at 8 o’clock.

Periodic Interval of Time [p. 153]

A sparse set of time intervals that describes periodic events with a duration, e.g., every Friday morning from 08:00 09:30.

Interval (IVL) [p. 166]

Also called "range". A continuous subset of an ordered type. Intervals are expressed by boundaries of the base type. Boundaries may be undefined.

Annotated Information (ANT) [p. 171]
: Whenever a sender feels that "there is more to say" about a data element, the annotation structure can be sent that contains the data element together with some free form annotation. The annotation is meant to be interpreted by humans.

History (HIST) [p. 173]
: Generic data type that allows the history of some data element to be sent. A History is a list of History Items [p. 174] .

History Item (HXIT) [p. 174]
: A History Item can be used wherever a validity time (effective date/time, expiry data/time) is essential part of some data. Used primarily as the element of a History [p. 173] .

Uncertain Discrete Value using Probabilities (UDVP) [p. 177]
: A discrete value and an associated probability for that value to apply in a given context.

Non-Parametric Probability Distribution (NPPD) [p. 178]
: A collection of Uncertain Discrete Value using Probabilities [p. 177] to specify a probability distribution.

Parametric Probability Distribution (PPD) [p. 180]
: Contains mean, standard deviation and also a distribution type plus its parameters. This is useful, for example, to specify "precisely" the accuracy of a measurement or to specify results of clinical trials.

Uncertain Value using Narrative Expressions of Confidence (UVN) [p. 187]
: A discrete value and a narrative expression of confidence for that value to apply in a given context. Those "narrative expressions" are keywords, such as "approximately", "probably", "likely", "slight chance of", etc.