# FHIM modeling

Contents

[Introduction 1](#_Toc379801713)

[Conventions 1](#_Toc379801714)

[Patterns 5](#_Toc379801715)

[Explicit Properties and Coded Types 6](#_Toc379801716)

[Entity/Role pattern 6](#_Toc379801717)

[Absence 7](#_Toc379801718)

[Laterality 7](#_Toc379801719)

[Concept post-coordination 8](#_Toc379801720)

[Appendix: Kinds of models 9](#_Toc379801721)

[Conceptual models 9](#_Toc379801722)

[Logical models 10](#_Toc379801723)

[Physical models 12](#_Toc379801724)

[The FHIM Kind of Model 12](#_Toc379801725)

[Appendix 1: HL7 RIM 14](#_Toc379801726)

[Appendix 2: NEIM 14](#_Toc379801727)

[Appendix 3: ? 15](#_Toc379801728)

## Introduction

This document is a guide to styles and conventions used in FHIM modeling to help modelers make choices consistent choices.

## Conventions

**Packages**

As the healthcare domain is very large, it is convenient to partition the domain into smaller portions for consideration. These portions correlate with UML packages, in which related classes will be stored. Additional packages are used to store data types (Datatypes) or shared constructs (Common).

Packages are an artificial grouping of portions of the healthcare domain; the federal information model is to be considered the union of the packages. Packages also form the basis of version control; that is, users will check out and check in work at the package level.

While package boundaries are logically void of meaning and have no effect on the semantics of the model, they do affect publication. When a package is published, it may “depend” on other packages—if it contains classes from other packages. The “Common” domain is designed to minimize these dependencies by collecting commonly reused elements in one place: many domains will depend on Common, but they should not depend on much more.

The reason for minimizing dependencies is that for both document publication and code generation, it is helpful to be able to implement part of the model without having to pull in other parts that may be irrelevant or incomplete. It is possible to select parts of packages for publication, but it is a manual process that may be error-prone: it’s best to design packages that minimize the amount of processing, where possible.

We adopt the following conventions:

1. Packages must have simple generic names (e.g., “Laboratory”). We do not add contextual or technical annotations.
2. Package names may not contain spaces, underscores, or other punctuation.
3. Each package shall be saved in its own file. The name of the file shall be identical to the package name. In Rational Software Architect, the primary model file is given the .emx extension (e.g., “FHIM.emx”). Domain packages are saved as linked sub-files (model fragments). In RSM, these sub-files have .efx extension (e.g., “VitalSigns.efx”).
4. A package is a namespace, and two packages, technically, may contain classes with identical names. However, in the interest of clarity, all FHIM class names must be unique in the FHIM.
5. A package may contain other packages, in cases where a subdomain is evident.
6. A package contains a main diagram with the same name as the package. Additional diagrams may be in the package: glyphs for these should be embedded in the main diagram so that it is possible to get an idea of the scope of the package from a single diagram, however lacking in detail that diagram may be.

**Names**

Package, class, and diagram names are upper camel case, capitalizing each word (e.g., “ReportableResult”). Property names are lower camel case, skipping capitalization on the first word (e.g., “observationMethod”). Stereotype names follow the convention of the stereotyped element.

In general, property names do not indicate the type of the element; there are some exceptions:

* “Date” is fundamental to the semantics of a date element, so “date,” “time,” or “dateTime” is prepended to names of temporal elements.
* Identification is fundamental to the semantics of identifier elements, so such elements will usually contain the word “identifier”: e.g., “patientIdentifier.” This is not always the case: “SSN” is clearly an identifier, but its name is clear without the suffix.
  + To do: Standardize on “identifier” or “id”
* Boolean values are semantically the answers to yes-or-no questions and are named with a verb, usually “is”: e.g., “isDeceased.”
* Strings should be named for the semantic content of the expected string, and need no suffix.
* Quantities should be named for the semantic content of the expected quantity, and need no suffix. Names may specify the quantity (“containerDiameter”) but should not specify units.
  + To do: harmonize barrierDeltaQuantity, bottomDeltaQuantity; perhaps as barrierDeltaDistance, bottomDeltaDistance.
* Codes should be named for the semantic content of the expected coded value, and need no suffix ; e.g., not "genderCode," but "gender."
  + Sometimes, a coded property is a classifier that is so fundamental to the semantics of the class that it is difficult to invent a name; in these cases, it is common to use “code.” Alternatives may be unsatisfactory: “kind,” “type,” “class,” and “category” work for the first classifying property, but become confusing if there is more than one. Fort this reason, it is optimal to identify the property that the code classifies if at all possible. If this is not possible, we recommend “kind,” as “type” and “class” already have specific meanings in the modeling domain, and “category” is vague enough to encompass less intrinsic qualities.
    - To do: address Specimen.specimenType: this is an amalgam of substance and source site; since source site is recorded elsewhere, this should be Specimen.substance. Note: even the SNOMED modeling for this remains ambiguous, including site where the substance may not be known (“esophageal specimen”) That should probably be “specimen not otherwise specified” with a source site of “esophagus.”
    - To do: address LabTest.code: these are LOINC codes, so it’s tempting to use the “code” term to identify them. However, LOINC itself is an acronym for “Logical Observation Identifiers, Names, and Codes.” Identifiers and codes are just special kinds of names, so we can assert that this property is the name of the test, though it’s an encoded name. Propose “LabTest.name.

Property names should not include class names. While this is a common practice in relational modeling, class models typically identify properties by path, so the name will include the class name anyway. The name of a Patient is stored in Patient.name: if the attribute were called patientName, then it would be addressed as Patient.patientName.

Names should almost always be singular, as almost any class or property, even if it has a multiple upper cardinality, will have a lower cardinality no higher than one. Exceptions are made for terms that are so habitually ingrained as to render a change confusing, e.g., the “Orders” package.

**Optionality**

The model will typically support optionality for most properties. This is to allow the development of a variety of use cases, some of which may have very modest requirements. As a result, very few properties in FHIM packages are required, and fewer are mandatory (i.e., required and not nullable). The idea of a patient, for instance, seems meaningless in the context of medical systems without a patient identifier, and it is mandatory. But usually, even when a property seems fairly fundamental to a class, there may be cases where it is unknown.

The diagrams generated by the modeling tool show cardinality as two numbers separated by two periods (e.g., “0..1”). The first number represents the lower bound on the allowed cardinality, the second the upper. If the cardinality is unrestricted, the number is replaced by an asterisk. If the upper and lower limits are the same, the limit is shown once, without periods.

The diagrams have a default cardinality of 1:1: if cardinality is not specified, that’s what it is.

**Null values**

(see appendix 4 for more options)

The FHIM supports null values for cases in which a scenario may require a property that the partner cannot provide. It effectively makes a required property not required, but it establishes an expectation, without which virtually all properties would have to be optional.

* Each property may have a nullable or non-nullable type. The only distinction is that it is valid to send null instances of the former: no specific implementation of this element is specified. It is expected that HL7 V3 implementations will, where not otherwise specified, use the most general null flavor, “no information” (NI).

In addition to the general concept of “null”—no value provided—HL7 recognizes several “flavors of null,” which add metadata about why a value may be null. This feature uses a property inherited by every data type, so it can create a lot of overhead, both in type instantiation and value checking. We hold that in such cases this metadata can be modeled explicitly. “Last menstrual cycle” might be accompanied by “not applicable”; “allergies” should support “unknown.” The FHIM data types do not require features to carry this information.

There is one case where metadata can be important often enough to merit a data type feature: the case of coded elements that permit “other” values. Where specification designers permit the use of codes outside of an enumerated set, it may not be necessary to indicate when this option is chosen. It may be necessary, however, to explicitly state that an instance invokes the “other” option in order to distinguish “other” instances from errors so that, if the instance does not assert that it is “other,” then the receiver can validate the value against the value set. If this is necessary, then we have a requirement:

* Coded properties will also support the indication that the sender has chosen a value “other” than those recommended by the specification.

**Requirements Tracing**

Each data element in the FHIM should document its source, so that future discussions about whether it is needed or what it means can be grounded in actual usage.

The format may depend on the source. HL7 V2 elements are commonly identified by segment abbreviation and element number within the segment, e.g., “OBX-5” for observation value.

The method of documenting these sources has not yet been determined. There are several options:

1. Include the source in the “documentation” of the field. This makes the definition a bit messy, and it would be difficult to audit programmatically, but it’s very convenient.
2. Tagged value
3. Stereotype
4. Relationship to a “Requirement” classifier

## Patterns

**Properties & Classes**

It’s often obvious when a data element should be a property of a class rather than a class of its own—but not always. The criterion people use is whether the property is intrinsic to the class. But “intrinsic” may not always seem to be a usefully concrete criterion, so there are some heuristics for deciding:

1. If an element has structure of its own, it will need its own properties, and should therefore be a class. A person’s citizenship may have a date range, so the role of Citizen is a class separate from Person.
2. If the element is associated with the proposed class in most or all proposed use cases, it may be a property; if it’s often not relevant, it may belong in a separate class. The UniformedServicesPerson class holds rank and grade in a class separate from Person because that information is not expected to be prominent in many use cases.
3. If the element has multiple cardinality, it’s a candidate for a class. Properties can have multiple cardinality, so this isn’t a strong rule, but it may indicate a possible need to treat the element separately.
4. If the element is significantly more volatile than the class, it may indicate intrinsic difference. People’s names change periodically, but not often: Person.legalName is a property. Addresses, however, can be expected to change at least a handful of times for every person.
   1. To do: Need a better example: Addresses are properties in FHIM. And need to clarify where we think the boundary is.

**Data Types**

FHIM data types are conceptual; i.e., they communicate semantic requirements in a way that subject matter experts can confirm: names are text; gender is coded. They do not necessarily contain all of the information that might be required for a particular implementation.

FHIM data types come from two places: UML primitives, and a simplified subset of the HL7 datatypes. These types must carry enough instance data to support a variety of implementation types—including HL7 V2 and V3. The FHIM types can only simplify source types where a) the source type has data that can be inferred from other data or where FHIM limits use cases to use only the selected features. No FHIM-constructed specification, for instance, will support code translations (unless and until we find a use case that requires them, at which point we will either modify the domain or the data type).

In general, complex types are regarded as types, not data types. This is partly to support subject matter expert review: if the address is explicitly modeled on the diagram, its assumptions are clear, whereas if that structure is embedded in the data type tag of a property, they are not.

### Explicit Properties and Coded Types

Often, a class may contain properties that are very similar, differing only in, say, a context of use. For instance, a Person may have several phone numbers, which may differ only by mode (e.g., cell or pager). When discussing requirements with domain experts, these are likely to be captured as separate requirements, and specifying them separately ensures that they remain visible to those experts.

A modeler, with an eye on maintaining consistency and minimizing repetition of things to be maintained, might create a new class with both a telecommunications address property, to capture the number, and a type property, to indicate whether it is a cell, land line, or pager. This pattern will capture the same information as the specifically modeled pattern, but it does so in a way that maximizes parallelism and, in the bargain, guards against change. If a new type of device is developed, codes for the type must be updated, but the model need not be changed, and the impact to downstream assets like specifications and application code is minimized.

The trade-off is between the clarity to stakeholders that a conceptual model offers and the abstract pattern that encourages parallelism and reduces maintenance one expects in a logical model.

To do: decide on style and a justification for choice

### Entity/Role pattern

One thing the HL7 Reference Information Model does well is separate things as they are from the roles they play. We don’t assign patient identifiers to people; we assign them to the patient roles they play. In this way, a person might play patient roles at many clinics, and each role might have different properties, but they don’t interfere with one another. The FHIM adopts this pattern where appropriate.

### Absence

In some cases, the absence of a thing is significant. If, for instance, a patient has no allergies, then simply not asserting that the patient has any particular allergy might leave a subsequent user of the data in question as to whether the patient actually has no allergies or the data was simply not collected. How, then, is this ambiguity to be resolved?

Several patterns meet this need: [To do: choose one. C-CDA currently discussing again (2/14)]

* Sufficiency: the position that the model is sufficient. This approach may work in specifications, where a sender is required to ask about allergies so that an empty “allergy” section can be asserted to mean “no allergies,” but the FHIM cannot constrain its use to such a scenario. There may be systems that record information about patients for whom allergy information has not been recorded, and it is not possible to infer whether the information meets such a criterion without explicit indication.
* Negation: instantiate an allergy, but negate it, asserting that “this notional allergy negates the existence of any undocumented allergy.” This is the approach followed by the CDA Consolidation project, and it leverages the semantics of the HL7 RIM. However, although participants following the CDA Consolidation implementation guide have an unambiguous reference, the semantics regarding what exactly is being negated are not immediately obvious to anyone not explicitly bound by that guide.
* Encoding: include a “no allergies” value in the allergy value set. This approach offers clearer semantics than negation, and it is a popular pattern as well. The problem is that it involves including a value in the value set that is not congruent to the other values. The value set is “kinds of allergies”; “none” is not a kind of allergy, and any reasoner trying to classify patients by allergy is likely to have a problem with it.
* Explicit question: include a separate property to represent the presence or absence of allergies. This property can be True (there is at least one allergy), False (there are no allergies, to the best of the recorder’s knowledge), or Null (the absence of recorded allergies does not imply that the recorder asserts that there are no allergies; this is unknown). If the value is True, the specification may be expected to require at least one allergy record. This seems to be the safest option, though by implicitly duplicating the positive assertion, it makes contradiction a possible concern.

### Laterality

We follow the SNOMED CT position that laterality is used to distinguish bilaterally symmetrical structures, e.g., “left kidney,” and not for relative positions, e.g., “left side of kidney.” For the second case, we use the term medial or lateral with respect to the structure’s position in the body: e.g., site “left kidney,” modifier “lateral aspect.” In addition, we use the precoordinated lateral concepts for the first case, e.g., “left kidney.” These policies *almost* make the terms “left” and “right” unnecessary. But they remain needed for two cases: where the bilaterally paired structures don’t have precoordinated concept identifiers, and where bilaterally symmetrical structures have two lateral aspects, e.g., site “lateral aspect of face,” modifier “left.”

As a result, the FHIM records a body site in three possible elements:

* Body site. This may be sufficient (left kidney, entire mitral valve).
* Body site modifier. This will often be necessary (e.g., lateral aspect, distal, superficial).
* Body site laterality. Ideally, only necessary for lateral aspects of singular structures (e.g., “lateral aspect of face”), but probably necessary for a period of time for bilaterally symmetrical but as-yet-unprecoordinated concepts, such as “entire cortical lobule of kidney, left”

### Concept post-coordination

Post-coordinated expressions are a very powerful tool for expressing semantics either not anticipated by a specification developer or left unspecified due to asset management concerns. It’s easier to manage ten concept identifiers that can be combined into 200 post-coordinated concepts than to manage the 200. Proponents of post-coordination also correctly point out that post-coordinated concepts are not fundamentally different from those identified explicitly in a model or value set; they are only modeled using a different formalism.

Still, this approach concerns us for two reasons. First, the clear majority of implementations use database technology that does not parse post-coordinated expressions. This is one reason the “Term Info” project—chartered to provide guidance on using SNOMED CT concepts in HL7 V3 models—provides not only a “best” way to do things (usually involving expressions), but also an “acceptable” way (never involving expressions).

Second, the very flexibility of expression that post-coordination supports begs the question of whether a recipient will understand it. Even if the recipient can parse an expression, if the value has not been anticipated by the system designers, the system won’t know what to do with the resulting information.

The FHIM does not support post-coordination at this time.

## Appendix A: Kinds of models

There are many communities that practice data design, and each has its own idiom for discussing the objects of its concern. Often, they will use the same terms to mean similar but not identical things. Any effort to construct definitions that are satisfactory to more than a few practitioners will show how various their meanings are. We adopt the following definitions, with the hope that these explicit definitions will provide a reference point by which these communities of practice can better understand one another.

One set of terms common to many communities is that of conceptual, logical, and physical models. The characteristics of each of these model types derive from its purpose.

### Conceptual models

The purpose of the **Conceptual** model is to document requirements in a way that is clear and perspicuous to the domain subject matter experts. The conceptual model might be presented as a textual description: the only reason to use UML at this stage is that it is easier for many people to comprehend the diagram than long lists of very precise statements. Compare the following representations:

|  |  |
| --- | --- |
| A person is a thing  A person always has an id  A person may have more than one id An id is an identifier  A person always has a birth date  A birth date is a date  A name is a thing  A person always has a name  A person may have more than one name  A name always has a type  Name types come from a defined list  A name always has a “last” part  A “last” name is text  A name may have a “first” part  A “first” name is text  A name may have a “middle” part  A “middle” name is text  A name may have a title  Titles come from a defined list  A name may have a suffix  Suffixes come from a defined list | Figure |

On the one hand, the text is unambiguous. No special training is required to make sense of it, whereas the picture adopts certain conventions for the representation of knowledge, and these conventions must be learned to be clear. However, after a small investment to learn these conventions, figure 1 is much easier to scan, both for initial comprehension and to identify problems. The format makes certain questions hard to miss. You might leave out the sentence “Titles come from a defined list” and not notice, but the uniformity of the datatype specifications in the model diagram would make the gap obvious.

The purpose of this model is to represent information requirements clearly, so that domain experts and implementers can agree on those requirements. Some experts may actually prefer the textual presentation; some may prefer other graphical presentations (e.g., “mind maps”). The use of UML has a slight cost in its adoption of specific conventions that must be learned. In its favor, these conventions are uniform: they only have to be learned once, and you might expect them to be understood by more people—without additional explanation—than conventions based on any particular idiosyncratic approach, however lucid that approach may seem to its author. Furthermore, these conventions are uniform, easily adaptable to the next (“logical”) phase, and, perhaps most important, they provide a framework to minimize the chances of missing information, as in the “title” example above.

### Logical models

Figure

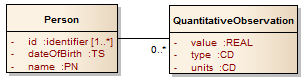
The second type of model is “**logical**.” This does not imply intellectual superiority to the conceptual; rather it means that the model follows a particular logic of construction. In the person example above, it would be quite possible to create a “Person Name” datatype, model it like the Person Name class in the diagram, and simply give the Person class a “Name” property with that type, as in figure 2. It would have the same meaning: neither way is any more “correct” than the other. Going the other way, you could make each name part (first, middle, last) its own class. There are infinite ways to model a domain, and in the conceptual layer, it makes no difference which one you choose. You can even use different paradigms in different areas of the model, if necessary. As a common proverb in the modeling world has it, “All models are wrong; some are useful.”

But the logical model is designed to communicate requirements to the technical team in a way that is not only unambiguous but, to the extent possible, consistent. The purpose of this uniformity is to support reuse. It’s hard to make a model homogeneous: it involves some abstraction, and homogeneity is absolutely not necessary to meet the requirements of any particular use. But a very important requirement is to be able to access information stored in one scenario for use in another. For instance, you may model blood pressure differently from temperature, but it would be useful for a vital signs application to treat them similarly: the logical model strives to make them similar enough for the application to make some generalizations. Whether the scenario simply involves two clinicians looking at data in the same format but in different applications, or it involves repurposing the information in an intelligent way for clinical decision support or research, we can support that reuse by putting the information into predictable shapes. In order to do so, the logical model adopts patterns.

One such pattern is the set of datatypes that the model uses. The conceptual model examples above also specify datatypes so that experts can agree on whether an element is, e.g., a code or text, but the logical model must do so to a greater degree of specificity. At the conceptual level, “coded” simply means that there is a set of values that are valid, and that at runtime, a user must choose one of them. At the logical level, there is a more formal understanding of what exactly must be recorded—just the legible text of the valid value? A unique non-semantic code that identifies it? The name of the system from which the code was drawn? The date at which it was drawn from that system?

A more striking pattern, to the casual observer, is the set of relationships classes are permitted to have. Example: one might, while capturing requirements around vital signs, prepare this (figure 3) conceptual diagram. (This example excludes many interesting properties in the interest of clear illustration.) The relationship seems clear enough. However, if one also wanted to represent diastolic pressure, or heart rate, or any number other measurements, the diagram would get quite crowded. This crowding would be much more severe if the diagram showed the many contextual elements that are required to ensure proper understanding of the concept. And this crowding would not simply be an inconvenience for the viewer of the diagram; it would make maintenance more difficult and error-prone, and, by complicating maintenance, it might also affect data quality, and thereby quality of care.

Figure

The crowding can be alleviated if the values are put in one place. In this example, we can rename SystolicBloodPressure to QuantitativeObservation and put any number of observations into the “value” field. Not only does this cut down on the clutter in the diagram, but it cuts down on semantic clutter in the system. Any application designer who wants to read information patterned in this way learns the pattern once and then always knows how to find needed information, with a consistent way to identify time, or location, or provider (another Person with a different Role), or device (another participation).

Figure

Figure 5 shows another pattern. This particular pattern, relating things like people to activities via “roles” and “participations,” is the fundamental pattern of the HL7 Reference Information Model, or RIM.



Figure

It’s not the only pattern one could use, but it is flexible enough to be used very consistently, and it forms the basis of the Clinical Document Architecture, one of the most widely adopted healthcare information specifications.

### Physical models

The third level of model is the **Physical**, also called the implementation layer. A physical model imposes design constraints that are unique to a particular implementation technology. In the relational database world, a physical model would specify platform-specific data types and lengths (“varchar 20,” e.g.), primary and foreign key values to support SQL queries, and indexes to speed up searching. A physical model is what is actually used by executable software to decide how to behave.

The beauty of the logical model is that it defines all of the requirements from the conceptual model, but it does so in a way that is predictable. Because it’s predictable, it can be used by automatic tools to generate derivative products, including, given enough information about a target physical platform, physical layer specifications. Given a sufficiently detailed logical model, no one ever needs to manually design a physical model—though it may be advisable to check the product to make sure the designer and the technologies share the same assumptions.

There are those who, impressed with the power of the logical model, hold that the conceptual model is not necessary. It is true that in some environments, it may be possible to document requirements and shape them into a logical model simultaneously, but such environments would be rare, characterized by small, closely knit sets of stakeholders, uniformly knowledgeable in both medicine and modeling practice. The set of groups who believe they meet these criteria is likely to be larger than the set of groups who actually do.

### The FHIM Kind of Model

The FHIM is a hybrid of model types. It looks much like a conceptual model, in that it does not specify classes that are present purely in order to articulate a logical pattern. Most classes have properties that seem inherent at the conceptual level, but too specific to be part of a logical design—livingArrangement for Person, for instance. These properties, if put in the RIM, would have to be converted into separate observations

However, the FHIM classes have UML stereotypes that map them into these patterns. The Person class, when transformed into a RIM-compliant artifact, will become an Entity of type Person, with an associated livingArrangement observation.

While the FHIM looks like a conceptual model, it does use some patterns. But they are really meta-patterns, designed to ensure that FHIM classes can be transformed into a number of specific logical patterns (currently the HL7 V3 RIM and NEIM). The rules for these transformations are embedded in UML stereotypes, which the use cases will use to convert, e.g., the FHIM person class into the CDA person entity, with associated observations for things like livingArrangement. The specifics of these patterns, and their implications for conceptual modeling, are documented in the Appendices.

The meta-pattern that the FHIM follows is largely inspired by the HL7 RIM. It does so partly to support implementation in V3, and partly because some general pattern was needed, and the RIM is familiar to many in the interoperability domain. As a result, when a printed class name includes its generalization (e.g, “Person (living entity)”), the generalization will usually be congruent to a RIM class.

## Appendix B 1: HL7 RIM

The RIM backbone asserts the following:

* Entities are related only to roles, both as playing entities (the one in the role) and as scoping entities (the one defining the role for someone else, e.g., as an employer).
* Roles may be related to other roles in Role Links. These are rare.
* Roles may be associated with Acts via Participations.
* Participations may be associated with Roles and Acts.
* Acts may be associated with Participations and, via ActRelationships, other Acts.

That is the foundational pattern: any class being converted into an HL7 V3-based specification must be one of these classes, must make its properties conform to the chosen class’s properties, and can only participate in the relationships outlined above.

CDA follows a constrained model based on the RIM.

A class in the FHIM intended to be used in a RIM-based specification must state conformance to a profile that maps it to a RIM class or a set of RIM classes. In some cases, this may be trivial: if a class contains only properties that can be mapped to the properties of a single RIM class, and all of the class relationships conform to allowed relationships for the RIM class, the profile will simply be a mapping of property and relationship names.

If, however, the FHIM class contains properties not supported by a RIM class, a more complex profile will be necessary. For instance, the Patient class in the FHIM contains some properties (e.g., identifier) that map neatly to the Patient clone of the Role class in the RIM. But it also contains an indicator for the patient’s election to be an organ donor. The RIM won’t support that in the Role class, so something else must be done. In this case, the profile should indicate that Organ Donor Indicator should be modeled as an observation class with a specific question code (e.g., LOINC code 44788-8, organ donor) and a value data type of Boolean, and that the Patient role participate as the subject of that observation.

A set of HL7 RIM profiles are available for most FHIM classes; as new classes are added, they can be assigned to existing profiles if those profiles support the new requirements, or new profiles can be composed.

When a use case is defined that requires the use of the RIM, part of the work, in addition to identifying required elements and adding use-case-specific constraints, is confirming that the stereotypes are applied correctly. A class containing three observations, for instance, should generate three distinct observation classes, with appropriate relationships among them.

## Appendix B 2: NEIM

## Appendix B 3: FHIR

## Appendix C: Null value discussion

Documenting a discussion still in progress:

The FHIM supports null values for cases in which a scenario may require a property that the partner cannot provide. It effectively makes a required property not required, but it establishes an expectation, without which virtually all properties would have to be optional.

* Requirement: all data types shall support the option of specifying that a property is nullable, i.e., an element instance can be present without containing a value.

This specification does not stipulate any particular representation of “null”: this is an implementation decision. HL7 V3 artifacts will use null flavors for this feature; V2 uses empty strings delimited with double quotes (""); other platforms may simply omit the property or use other conventions.

In addition to the general concept of “null”—no value provided—HL7 recognizes several “flavors of null,” which add metadata about why a value may be null. This feature uses a property inherited by every data type, and it can create a lot of overhead, both in type instantiation and value checking. We may decline to adopt this approach in most cases, as most platforms do not have similar features, preferring to model such metadata explicitly where necessary.

However, we recognize three special cases.

First, it is sometimes useful to distinguish a positive assertion that a value is unknown from the mere absence of a value. That is, in addition to the “I’m not telling you anything about this value,” we want to be able to say “I considered this question and I don’t know the answer.”

* Requirement: all data types shall support the option of specifying that a value is unknown.

Second, it is sometimes useful to distinguish the circumstance where a value is not applicable, e.g., gender-specific observations for patients of the other gender.

* Requirement: all data types shall support the option of specifying that a value is not applicable.

And third, there is a need for the concept of “other” in cases where the domain of an element is not a formally defined, infinite set of patterned values (integers, times, decimal numbers), but a finite, enumerated set of concepts—i.e., coded concepts. In these cases, it is not uncommon for users to need to express concepts not anticipated by the authors of the enumerations, where specification designers permit this latitude.

The first step in supporting these cases is to indicate, in the binding of the property to the value set, that unspecified values are allowed. This could simply be another specialization of the coded type; there is no additional instance data to record, so there are no new properties to define for the data type. Instances using “other” concepts would look just like instances using recommended concepts; the difference would only be detected when validating the coded value and finding it not present in the value set.

It may be necessary, however, to explicitly state that an instance invokes the “other” option in order to distinguish “other” instances from errors. If the instance does not assert that it is “other,” then the receiver can validate the value against the value set. If this is necessary, then we have another requirement:

* Requirement: coded data types shall support the option of specifying that a value is not from the specified value set.

Whether this is a requirement or not, we assume that the coded value would be modeled in the same properties as a value from the specified value set, and that there is no need for a parallel set of code/system/text properties to support these “extensional” values.

This leaves us with up to four values that could be considered flavors of null—no information, unknown, not applicable, and other—providing a glimpse into the thought process that resulted in the flavors of null design in the HL7 RIM. However, note that they are not all that similar.

* No Information: Null
* Unknown, Not Applicable: Null, plus metadata about why it is null
* Other: Metadata about a value that is not null.

If we support Null in the manner described above, then the additional requirement is to support metadata about a particular element, whether null or not.

There seem to be 4 general solutions to this requirement:

|  |  |  |
| --- | --- | --- |
|  | Model property metadata in the information model | Model property metadata in the data type |
| Model property metadata as individual properties | A  Include properties in the specification where metadata is required.   1. This is a generally consistent solution, though it may introduce a level of detail that some find inappropriate to a conceptual model. 2. This approach will require a decision about whether these metadata properties can be included as siblings to their target properties, with the semantic relationship evident only from the name or description, or should be broken out into classes to associate the two values. The latter approach verges on quadrant B. | B  Provide each type with properties to indicate whether any of the metatdata applies.   1. This path may create a proliferation of types, and many of the types will have to re-define properties already defined elsewhere in order to avoid multiple inheritance (unless we can do it with stereotypes). |
| Model property metadata as one coded property | C  Include a coded property in the information model to capture required metadata.   1. This approach faces the same “property or class” decision as quadrant A, and has the same options as quadrant D. | D  Include a coded property in the data types to capture metadata.   1. This is the HL7 RIM approach, though we could leave “null” out of it, as well as many other more specialized values in the HL7 value set. |

And a fifth option, E: don’t model this metadata in the FHIM; defer it to the “use case/specification” process.

## Text requirements for coded types

We anticipate N possible scenarios for recording text in addition to the text defined in the value set as the appropriate representation of the concept.

1. The original text from which code was derived.
   * This case involves the conversion of information recorded as natural language (e.g., a clinical note) into structured data, whether manually or automatically.
2. Text displayed on screen for selection of code, if different from system-specified text.
   * This is semantically very similar to case #1, the only difference being the author of the text.
3. Qualifying text for an accurate but imprecise coded value.
   * This is a property that could be useful for many elements. We might expect use case specifiers to identify in the information model, or we might adopt a style in which this dimension is always available, built into the types.
4. The value for an “other” selection.
   * “Other” values should be communicated in the normal coded value attributes, even if there is no code available for a concept entered as text.

We propose that one “originalText” value will meet requirements 1 & 2.

Requirement 3 should be addressed by an information model element, in cases where qualification is required.

Requirement 4 should be met by the code and text elements used for values from the value set.

## Other code properties

Translation: Our specification is for the specified value. Participants may include specified values, or, if they use a different system and it is permitted, they may use that system and tag it with the “other.” We have no use cases at this time that specify the use of a primary value and an additional translated value.

Composition: Value set definitions may permit or prohibit expressions. This has no impact on the model, apart from an implementation need to address field lengths.

Rank, score, value: Ordinals may need magnitudes. One property should be able to support all three of these variant uses.

Order: This seems to be a display property that can be inferred from the value set specification, and does not need to be specified in the instance.