Flight Information Exchange Model

Modeling Best Practices

Executive Summary

The Flight Information Exchange Model (FIXM) is an exchange model capturing Flight and Flow information that is globally standardised. The need for FIXM was identified by the International Civil Aviation Organisation (ICAO) Air Traffic Management Requirements and Performance Panel (ATMRPP) in order to support the exchange of flight information as prescribed in Flight and Flow Information for a Collaborative Environment (FF-ICE).

This document specifies the procedures and best practices used to produce FIXM logical models. It is required that all logical models sponsored by the FIXM organization comply with these practices, to ensure that the models are complete, correct, and efficient. It is strongly suggested that all extension data models adhere to these best practices, to ensure proper interoperation with the FIXM core models.

August 15, 2014

Version: 3.0.0

NOTE: In support of ICAO FF-ICE, the content of the FIXM v3.0 reflects the continuous progress and evolution under discussion. Global convergence is expected to be achieved in FIXM 4.0.

Change History

Version	Date	Author	Reason for change
2.0.0	16/08/2013	MIT LINCOLN LABORATORY	FIXM v2.0.0 Release
3.0.0	15/08/2014	MIT LINCOLN LABORATORY	FIXM v3.0.0 Release

Table of Contents

Table of Figures

Table of Tables

References

1. FIXM Core Data Dictionary
2. FIXM Change Management Charter, version 1.0
3. FIXM Strategy, Version 1.0, 07/02/2014

1 Introduction

1.1 Purpose of the Logical Model

The FIXM Logical Model is the crucial intermediate step between the domain oriented concepts of the FIXM Data Dictionary (FIXM DD) and implementation artifacts such as the XSD (XML Schema Documentation) schemas and related documentation. The logical model adds structure and implementation information to the domain concepts and presents the resulting information in visual form as a UML (Unified Modeling Language) class diagram. The UML class diagram is a schema-neutral format that can be understood and reviewed by both domain experts and implementation engineers. The FIXM Logical Model is used as a common language for communication among domain experts and implementation engineers. Finally, it is used to generate the FIXM XML Schemas that define the canonical FIXM XML formats.

1.2 Relation to the FIXM Data Dictionary

With respect to the FIXM DD, items of the FIXM Logical Model fall into three categories:

1. Items derived from the FIXM DD: This class of item is a direct mapping from one or more data dictionary entries and contains information about a specific aviation (or related) domain concept. Examples are: Communication Capabilities, Last Contact Radio Frequency, and Beacon Code.
2. Structural objects: This class of item is used to organize derived items, but (usually) has no direct correspondence in the data dictionary. Examples are: Flight En Route data, Dangerous Goods Package, and Flight Emergency.
3. Basic objects: This class of item represents basic data types used to represent data elements from the FIXM DD, but have no direct correspondence to the FIXM DD. Examples are: Free Text, MultiTime, and Aerodrome Reference. Many of these types are derived from the Aeronautical Information Exchange Model (AIXM) or Geography Markup Language (GML) objects.

1.3 Derived Artifacts

The FIXM Logical Model is a design and communication medium, but it is also a source for further automatic processing. The following artifacts are derived automatically from the FIXM Logical Model:

1. XSD Schemas
2. Graphical XSD Schema Documentation (Using Oxygen[1] or similar XSD tool)
3. FIXM Logical Model Analysis Spreadsheet
4. FIXM Logical Model-to-FIXM DD mapping Spreadsheet

In later releases, the FIXM Logical Model may be used to derive other artifacts such as data access objects, documentation, or simulations. This implies that FIXM should remain implementation neutral, with regards to the physical modeling language, as much as possible so that the model retains the flexibility to support a wide range of derivations.

[1] Synchro Soft Inc., http://www.oxygenxml.com/

Best Practice 1 - FIXM model to remain implementation neutral

1.3.1 XSD Schemas and Documentation

The FIXM XML Schemas are the primary artifacts derived from the FIXM Logical Model. They capture all the data present in the FIXM Logical Model and define the physical structure of the XML representation of FIXM. The FIXM XML Schemas are produced from the FIXM Logical Model by a schema generation tool.

Accompanying the FIXM XML Schemas is an HTML (Hypertext Markup Language) representation of the schemas, prepared using the Oxygen schema design tool. These diagrams are a suitable reference for application development engineers who need to understand the structure and content of the physical model.

1.3.2 Model to FIXM Data Dictionary Mapping

A component of each FIXM delivery is a spreadsheet that illustrates the mapping of FIXM Data Dictionary elements to the FIXM Logical Model elements that implement them. This spreadsheet is used to trace data dictionary requirements to the FIXM Logical Model, to demonstrate coverage and to discover data dictionary elements that are not yet implemented in the FIXM Logical Model. Generating the mapping spreadsheet is partly automated by a name matching tool, but some hand matching is always required for cases where the matching tool cannot find a match or chooses an inappropriate match. Hence, the Data Dictionary-to-FIXM Logical Model mapping process is semi-automatic.

Best Practice 2 - Data Dictionary Items mapped to FIXM elements

1.4 Data Modeling Tools

1.4.1 Enterprise Architect

UML (Unified Modeling Language) is the standard representation of the FIXM Conceptual Model and FIXM Logical Model, and they are created and maintained using the Enterprise Architect Tool, version 9.0 or later.

Best Practice 3 - Enterprise Architect is the primary modeling tool

1.4.2 FIXM Data Modeling Workbench

Other actions required in development of the conceptual, logical and physical models are provided by the FIXM Data Modeling Workbench (hereafter, “Workbench”), a suite of utilities developed by MIT Lincoln Laboratory.

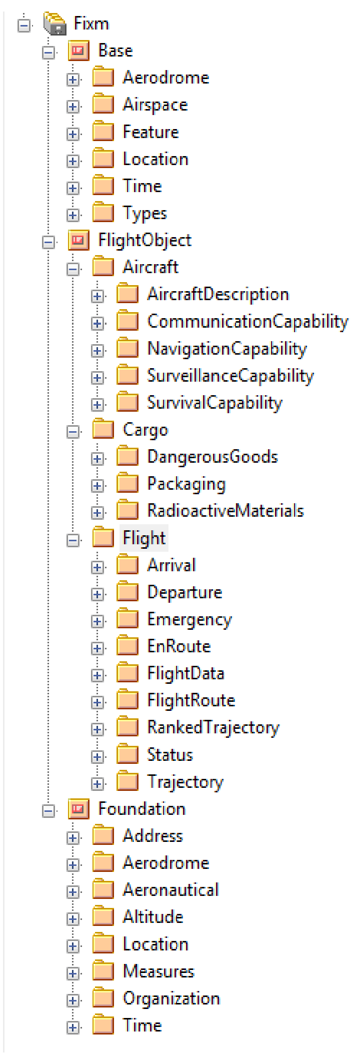
1.4.3 Oxygen Schema Development Tool

The Oxygen tool is used to generate the HTML documentation of the generated XSD schemas.

2 Modeling Elements

2.1 Packages

Packages are a logical subdivision of the FIXM Logical Model that allow readers and modelers to cope with its complexity by considering a few elements at a time. Package content is not chosen at random, but should reflect a unified theme, usually related to some aspect of flight management data, and this theme should be clearly stated in the package documentation. In general, a package should be limited to the number of elements and relations that can fit comfortably on a two-page Enterprise Architect diagram. As packages become larger than this limit, modelers should look for opportunities to divide them into logical sub-packages that can be managed independently. Figure 1 shows the FIXM package hierarchy as of this writing[2], taken from the Enterprise Architect modeling tool.



- Fixm
  - Base
    - Aerodrome
    - Airspace
    - Feature
    - Location
    - Time
    - Types
  - FlightObject
    - Aircraft
      - AircraftDescription
      - CommunicationCapability
      - NavigationCapability
      - SurveillanceCapability
      - SurvivalCapability
    - Cargo
      - DangerousGoods
      - Packaging
      - RadioactiveMaterials
    - Flight
      - Arrival
      - Departure
      - Emergency
      - EnRoute
      - FlightData
      - FlightRoute
      - RankedTrajectory
      - Status
      - Trajectory
  - Foundation
    - Address
    - Aerodrome
    - Aeronautical
    - Altitude
    - Location
    - Measures
    - Organization
    - Time

Figure 1 - FIXM Logical Model Package Hierarchy

[2] All figures shown in this document represent the FIXM Logical Model at the time of writing, but are purely for illustration and do not necessarily represent the current state of the model.

2.1.1 Foundation Packages

The package ‘Foundation’ and its sub-packages are reserved for elements shared with AIXM, and elements that depend directly on AIXM and GML types. As of this edition, the elements in the Foundation packages have been segregated from the rest of the FIXM elements, but not yet integrated with AIXM or GML.

2.1.2 Base Packages

The package ‘Base’ and its sub-packages are reserved for low-level FIXM elements that are shared by more than one logical model package but are not shared by AIXM or GML. In general, the Core packages will refer to and extend classes from the Base packages in preference to the classes of the Foundation packages, so the Base types provide an isolation layer to mitigate changes in the Foundation packages.

2.1.3 Flight Object Packages

The package ‘FlightObject’ and its sub-packages contain the elements that derive from the FIXM Data Dictionary, plus structural elements needed to organize those elements.

Best Practice 4 - Limit packages contents to manageable size

Best Practice 5 - All packages have a unifying theme stated in documentation

Best Practice 6 - Types derived from AIXM or GML appear in the Foundation package

Best Practice 7 - Commonly shared low level types appear in the Base package

Best Practice 8 - Entities derived from the Data Dictionary appear in the Flight Object package

2.2 Model Elements

Each data model contains a number of element types, each of which conveys its own piece of information about the concepts the model represents.

2.2.1 Diagrams

By convention, each package contains an Enterprise Architect diagram of no more than two pages, which illustrates the relationship among all the elements defined in that package, and any other items referred to by those defined elements. By convention, the diagram has the same name as the package.

Figure 2 illustrates a typical FIXM diagram, and will be used as an example in further discussions.

Best Practice 9 - Limit diagram size to two pages

Best Practice 10 - Give diagram the same name as its package

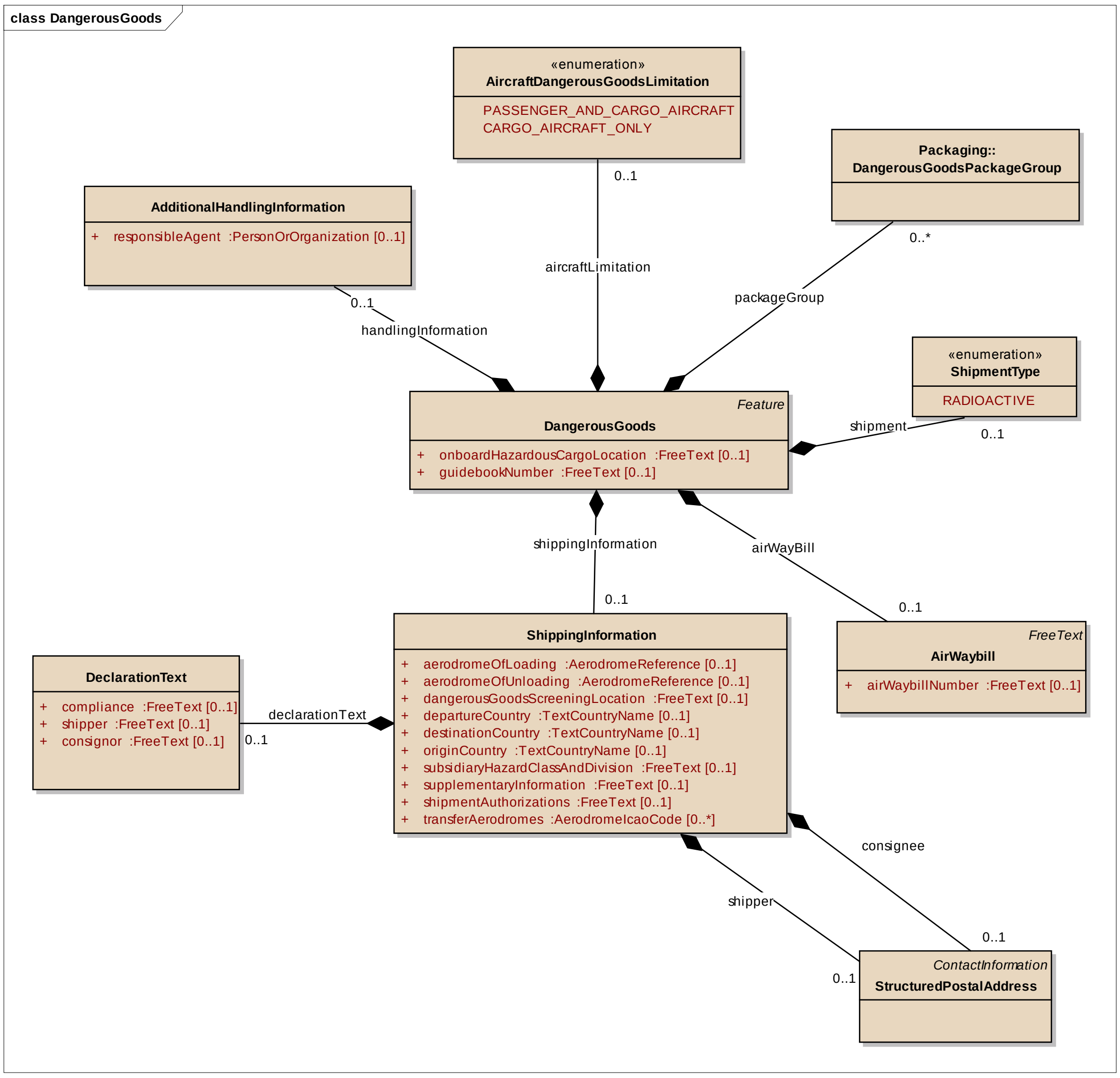

class DangerousGoods
«enumeration»
AircraftDangerousGoodsLimitation
PASSENGER_AND_CARGO_AIRCRAFT
CARGO_AIRCRAFT_ONLY
Packaging::
DangerousGoodsPackageGroup
AdditionalHandlingInformation
+ responsibleAgent :PersonOrOrganization [0..1]
0..1
aircraftLimitation
0..*
packageGroup
0..1
handlingInformation
«enumeration»
ShipmentType
RADIOACTIVE
Feature
DangerousGoods
+ onboardHazardousCargoLocation :FreeText [0..1]
+ guidebookNumber :FreeText [0..1]
shipment
0..1
shippingInformation
airWayBill
0..1
0..1
ShippingInformation
+ aerodromeOfLoading :AerodromeReference [0..1]
+ aerodromeOfUnloading :AerodromeReference [0..1]
+ dangerousGoodsScreeningLocation :FreeText [0..1]
+ departureCountry :TextCountryName [0..1]
+ destinationCountry :TextCountryName [0..1]
+ originCountry :TextCountryName [0..1]
+ subsidiaryHazardClassAndDivision :FreeText [0..1]
+ supplementaryInformation :FreeText [0..1]
+ shipmentAuthorizations :FreeText [0..1]
+ transferAerodromes :AerodromeIcaoCode [0..*]
FreeText
AirWaybill
+ airWaybillNumber :FreeText [0..1]
DeclarationText
+ compliance :FreeText [0..1]
+ shipper :FreeText [0..1]
+ consignor :FreeText [0..1]
declarationText
0..1
consignee
shipper
0..1
0..1
ContactInformation
StructuredPostalAddress

Figure 2 - Example FIXM Diagram

2.2.2 Classes

A "class" represents a logical object in the data model, or a collection of logical objects to be treated together. Classes are shown in the diagram as boxes and carry the following information:

- Name (e.g., TransferAerodromes)
- Optional stereotype (e.g., «enumeration»),
- Optional inheritance marker (e.g., AerodromeIcaoCode)
- Zero or more attributes representing data elements contained in the class.

Examples of classes from Figure 2 include "ShippingInformation", "Consignee", and "TransferAerodromes".

2.2.3 UML Attributes

A UML "attribute" is a name and datatype pair written inside a class rectangle, as shown by "onBoardHazardousCargoLocation" in the "DangerousGoods" class. UML attributes carry the following information:

- Attribute name
- Attribute data type
- Multiplicity (normally 0..1, 1..1, 0..* or 1..*, occasionally a range: 0..2). The default multiplicity of an attribute is 1..1.
- Visibility (must be public, signified by a "+" symbol)

By FIXM convention, UML attributes may only be used to represent items whose data types come from the Base or Foundation packages. UML attributes are primarily used to avoid cluttering the diagram with references to base types, and obscuring the references among Flight Object data types.

Best Practice 11 - Attribute datatypes are primitive, or from Base or Foundation packages

Best Practice 12 - Primitive attribute types allowed only in Base or Foundation packages

Best Practice 13 - Default attribute multiplicity is 0..1

Best Practice 14 - Attributes must have public visibility

2.2.4 Relations

The second type of property is the UML "relation" shown by a graphic link between two class elements. The "shippingInformation" relation between the "DangerousGoods" and "ShippingInformation" in Figure 2 is a typical example.[3]

Relations have the following characteristics:

- Containment mark: In the FIXM data model, this is always a black diamond on the source of the relation, and signifies that the source class contains the target class.
- Directionality: The direction of the relation is always from the source class (with the composition mark) to the target class. Arrowheads are not used to show directionality.
- Multiplicity: The multiplicity (usually 0..1, 1..1, 0..* or 1..*, occasionally a range: 0..2) is always associated with the target end of the relation. The default multiplicity of a relation is 0..1.
- Name: The name of the property is shown as the name of the relation, approximately at its midpoint, though this might be adjusted for clarity.[4]

By FIXM convention, UML relations are always used to connect a source class defined in the current package to another class defined in the same package or in another FlightObject package. Relations are never used to show a relationship to a class defined in the Base or Foundation classes. This

[3] It is often true that a relation has the same name as its target class element. This is usually because that name best expresses both the data and the relationship, but it is by no means required that the names correspond.

[4] This use of the relation name to represent the property name is non-standard UML, but is adopted because it causes less crowding of the diagram than the standard usage of attaching the name to the target end, along with the multiplicity.

convention is intended to minimize cluttering the diagram and obscuring references among Flight Object data types.

Best Practice 15 - All relations are containment composition

Best Practice 16 - Relation connectors do not show directionality arrowhead

Best Practice 17 - Relation names are attached to the connector

Best Practice 18 - Multiplicity is attached to target end of connector

Best Practice 19 - Relations refer to classes in the same or peer packages

2.2.5 Cross Package References

Most relationships in FIXM diagrams relate classes within the same package, as with "DangerousGoods" and "AirWaybill" in Figure 2. However, some relationships cross package boundaries to reference classes from a different package, as between "DangerousGoods" and "DangerousGoodsPackageGroup" in Figure 2. This is a legitimate use of relationships, because packages are artificial divisions of a continuous model space, but it is important to show the target object in the same UML diagram as the source object and the relationship. In Enterprise Architect, this is accomplished by dragging the target class onto the UML diagram before establishing the relationship. Cross package references are distinguished because the name of the target class is prefixed by the name of its containing package, as "Packaging::DangerousGoodsPackageGroup".

Best Practice 20 - Show both source and target of cross-package references

Best Practice 21 - Hide content of imported classes if needed to simplify the diagram

2.2.6 Copyright

Each diagram of the model should contain the copyright notice specified in Appendix A. This notice may be included as a note element, a text element, Artifact element or a link to a copyright artifact as shown in Figure 2.

Best Practice 22 - Include copyright notice in each model diagram

3 Best Practices

3.1 Naming

FIXM prescribes different naming conventions for the different UML elements:

- Packages and Diagrams
  InterCap[5] notation with an initial capital: FlightObject, DangerousGoods, etc. Diagrams are named identically to their containing packages.
- UML Classes
  Intercap notation with an initial capital: DeclarationText, TransferAerodromes
- Properties (UML attributes and UML relations)
  Intercap notation with an initial lower case: declarationText, transferAerodromes, etc.
- Enumeration values
  Enumeration values are written all in upper case, using only letters, digits, and the underscore character, as shown in "ShipmentType" of Figure 2.

Names in the FIXM data model are often taken directly from corresponding entries in the FIXM Data Dictionary, but this correspondence is not required. In general, modelers should use names that are long enough to accurately express the concept defined by the element, but should guard against needlessly long phrases. In Figure 2, the name "AircraftDangerousGoodsLimitation" is already at the limit of a usable name.

Best Practice 23 - Name characters limited to upper and lower case, digits and underscore

Best Practice 24 - Use InterCap notation for all names

Best Practice 25 - Use starting capital for packages and classes

Best Practice 26 - Use starting miniscule for attributes and relations

Best Practice 27 - Use all capitals for enumeration values

Best Practice 28 - Names should be expressive of data content or relationship

Best Practice 29 - Names should not be of unwieldy length

Other general naming practices are:

1. Abbreviate only when a full citation produces an unwieldy name or an abbreviation is widely used across the industry
2. Choose industry standard words and phrases
3. Avoid abbreviations in names unless the term is a widely understood domain term and the alternative is much longer or less comprehensible
4. Choose the British (that is, from the Oxford English Dictionary) when there are alternative English spellings.
5. All names must be unique within their scope (even though Enterprise Architect allows duplicate names).
6. Use singular nouns unless describing an explicit list structure.
7. Use present tense of verbs unless the concept requires past or future.

[5] Intercap notation is often called "Camel Case" or "Embedded Capitals" notation.

8. For enumeration values, choose a short phrase over a single character code, (e.g., choose “HEAVY” over “H”) unless the code is a dominant standard in the domain and using a phrase would obscure the meaning.

Best Practice 30 - Avoid acronyms unless industry standard

Best Practice 31 - Avoid abbreviations except for very long names

Best Practice 32 - Choose industry standard words and phrases

Best Practice 33 - Choose British spelling when there are alternatives

Best Practice 34 - Names unique within scope

Best Practice 35 - Use singular nouns except for explicit lists

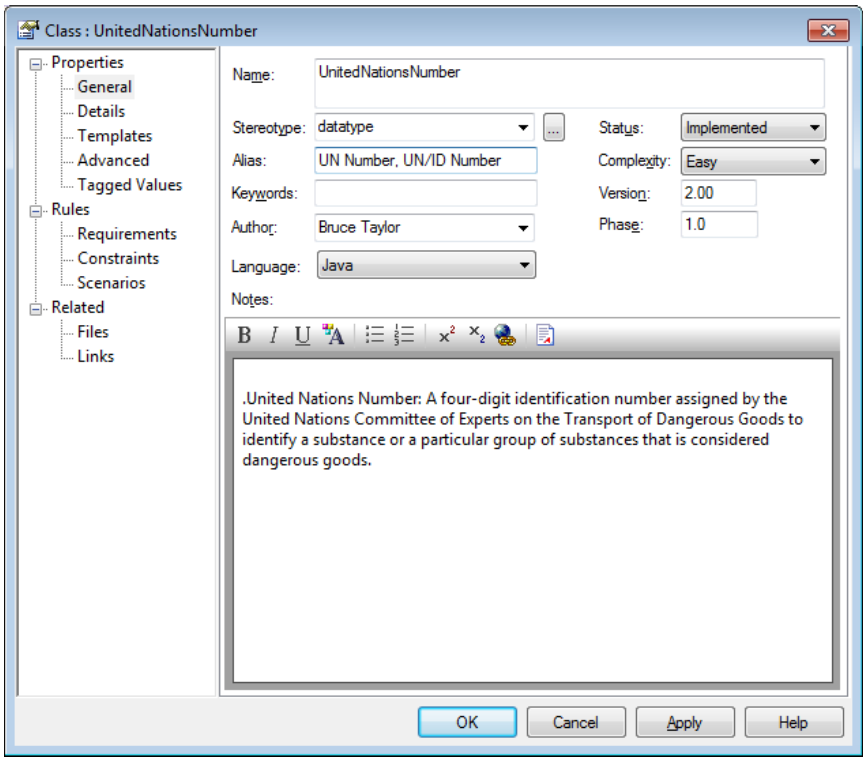
Best Practice 36 - Prefer present tense of verbs

Best Practice 37 - Prefer short phrases for enumeration values

3.2 Aliases

A FIXM entity, attribute, or relation may have, in addition to its primary name, a set of alias names, written as a comma separated list in its “Properties - General” tab as shown in Figure 3. These aliases are used only to capture additional concepts from the FIXM Data Dictionary, and to assist in name matching when mapping data dictionary entries to their implementations in the FIXM Logical Model.

Best Practice 38 - Use aliases to capture additional data dictionary mapping



Name: UnitedNationsNumber

Stereotype: datatype

Alias: UN Number, UN/ID Number

Keywords:

Author: Bruce Taylor

Language: Java

Status: Implemented

Complexity: Easy

Version: 2.00

Phase: 1.0

Notes:

.United Nations Number: A four-digit identification number assigned by the United Nations Committee of Experts on the Transport of Dangerous Goods to identify a substance or a particular group of substances that is considered dangerous goods.

Figure 3 - Use of Alias Names

3.3 Inheritance

UML classes may inherit from other classes, meaning that they share the contents and semantics of their parent, plus any other content that they define. In the usual way of object oriented data definition, a sub-type may appear anywhere that its parent type appears, but not vice versa. Inheritance may be shown in one of two ways, as illustrated in Figure 4: either by a directed arrow with an empty arrowhead or by showing the parent class type in italics in the upper right corner. Though both notations mean the same thing, the former notation is used to show when both parent and child are in the same package or in peer packages, and the latter is used when the parent is from the Base or Foundation packages (as in Direction).

Classes that are marked “abstract” are often used as the base for inheritance, but can never be physically instantiated in XML form: only their concrete descendants can be instantiated.

Chains of inheritance (i.e., class A inherits from B, which inherits from C) are permitted and are frequently used, but true multiple inheritance (i.e., class A inherits directly from both B and C) is forbidden.

Using Enterprise Architect, it is possible to designate entities as:

- “root,” meaning that it cannot be descended from another entity.
  The use of root is forbidden, because it has a very limited meaning for data modeling.
- or as “leaf” meaning that it cannot be further derived.
  “Leaf” is typically used to explicitly prevent the further extension of a type, so that it cannot be generalized beyond its design goal. For example, a list structure might be defined to have a maximum length, and then made into a “leaf” so that the length cannot be overridden to make the list longer. It is anticipated that “leaf” is used only rarely and after careful consideration.

Best Practice 39 - Abstract classes are allowed

Best Practice 40 - Root classes are forbidden

Best Practice 41 - Leaf classes must be justified

Best Practice 42 - Do not use "Abstract" as a prefix of abstract classes

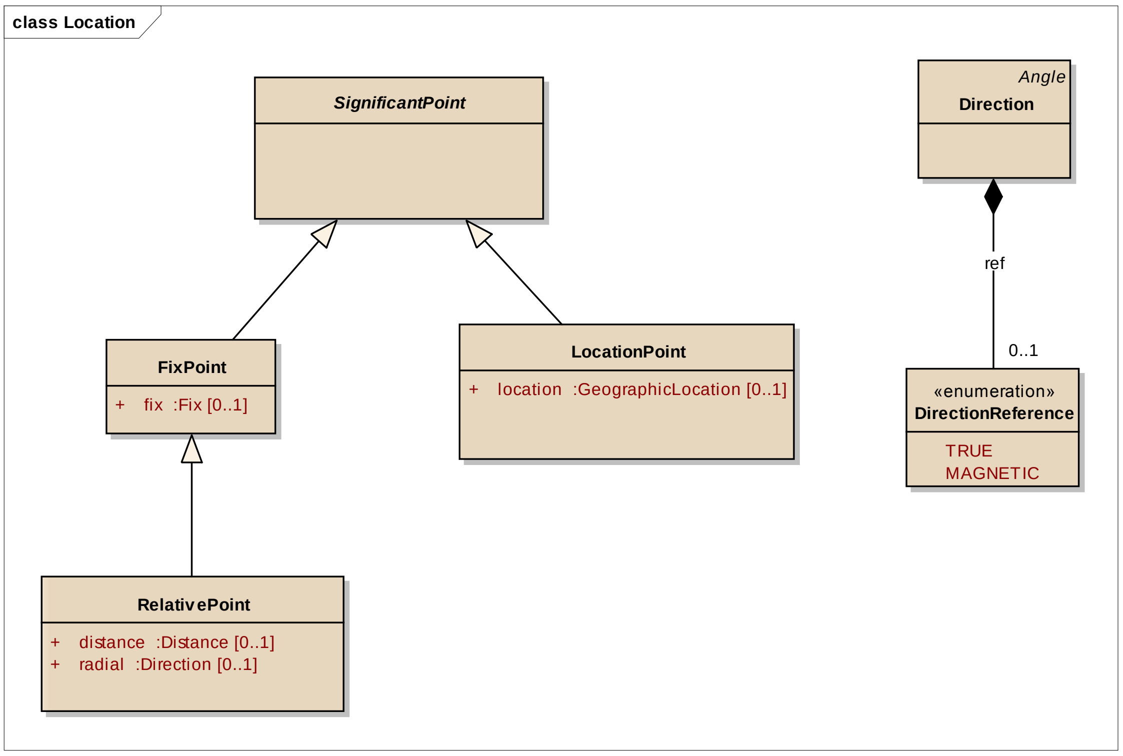


Figure 4 - Inheritance Notations

3.4 Data Types

UML properties always have a “datatype:”, for a UML attribute it is the name that occurs after the colon, as in type “Distance” of the “distance” attribute in Figure 4. For relations, the datatype is the type of the class on the target end of the relation, as in “DirectionReference” of relation “ref” in Figure 4. Primitive datatypes are available for use only in the “Base” and “Foundation” packages: within the “Flight Object” packages it is an error to declare an attribute of primitive type. The full set of allowed primitive types is:

- int
- string
- boolean
- float
- double
- long
- date
- time
- dateTime
- decimal

Best Practice 43 - Set of primitive types is restricted

Best Practice 44 - Primitive types used only in “Base” and “Foundation” packages

3.5 Constraints on Boolean Values

Because boolean elements have only two values: always “true” or “false”, their semantics depend on the context of their use and are notoriously subject to misinterpretation. Elements with only two states should be modelled as an enumeration, rather than a boolean. There are two types of enumerations to represent boolean values. Those that indicate a permanent state that was set and cannot be undone and those that are temporary states toggled by events.

For instance, if a cargo package is radioactive, it cannot later become not-radioactive. In such cases one enumeration value RADIOACTIVE is sufficient.

So, in Figure 2 the fact of a shipment being radioactive or not could have been modelled as a boolean attribute named “isRadioactive”, but the preferred practice, as shown, is to use an optional enumeration containing a single value: RADIOACTIVE. By convention, if this enumeration is present in the XML it indicates a true condition (i.e., the cargo is radioactive) and if it is missing it indicates a “false” value (i.e., the cargo is not radioactive).

A Second type of boolean enumeration contains binary values. For instance, a state of flight may become airborne as a result of takeoff but it can also become not airborne as a result of a landing after it has been airborne. Such event driven states are represented by binary enumeration.

In figure 5 such an enumeration is presented in the FlightAirborneIndicator class.

If no enumeration is supplied, it indicates that there is no information, or a state has not changed since it was last updated.

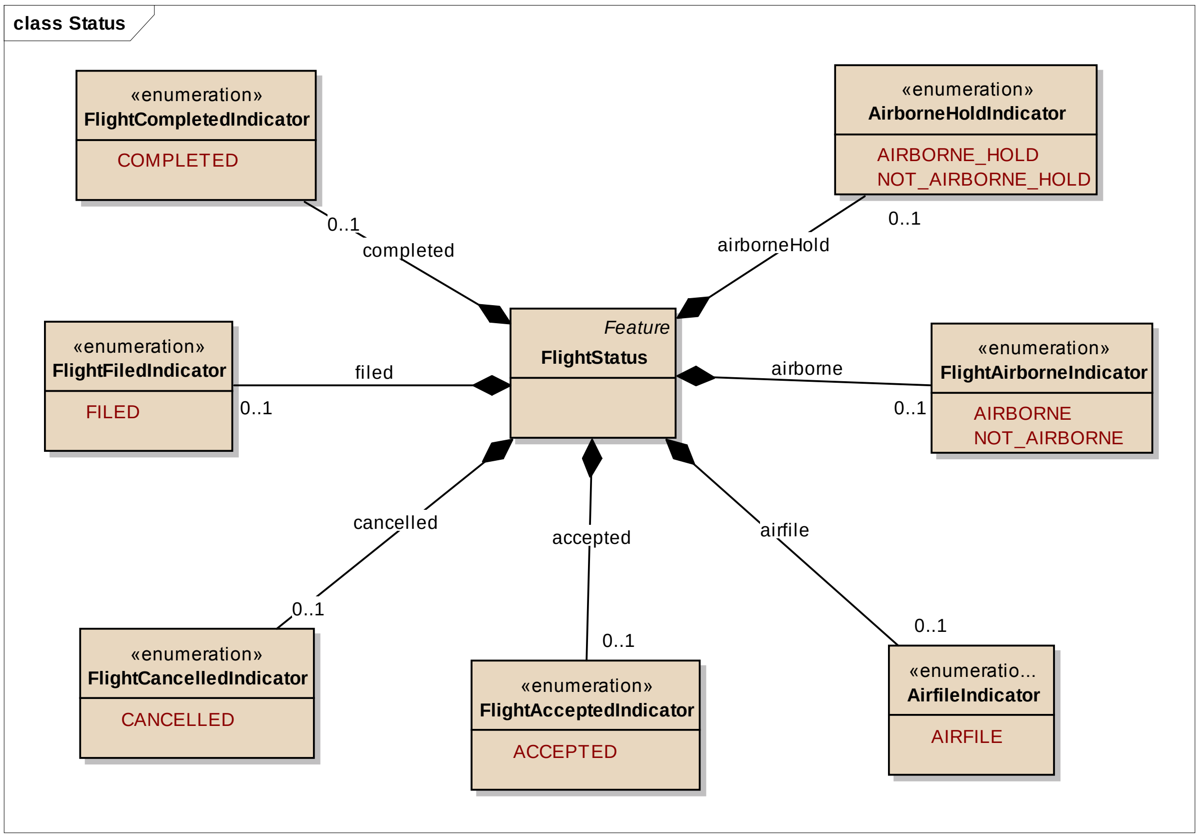


Figure 5 - Boolean Enumerations

Best Practice 45 – Use single valued enumerations to represent irreversible states

Best Practice 46 – Use double valued enumerations to represent reversible states

3.6 Stereotypes

Stereotypes are UML conventions that convey information about how a class or property is intended to be used. In FIXM, stereotypes are restricted to the following set:

- none
  classes from the FlightObject packages normally have no stereotype marker, except for the «choice» or «enumeration» stereotypes (see below).
- «choice»
  Represents a selection of exactly one of its component parts: a way of representing "either/or" logic. Figure 6 illustrates a choice class, and shows that attributes (otherColour) can be intermingled with relations (colourCode) as alternatives. The cardinality of alternatives must be 0..1.
- «enumeration»
  Represents a selection from a set of named string elements.

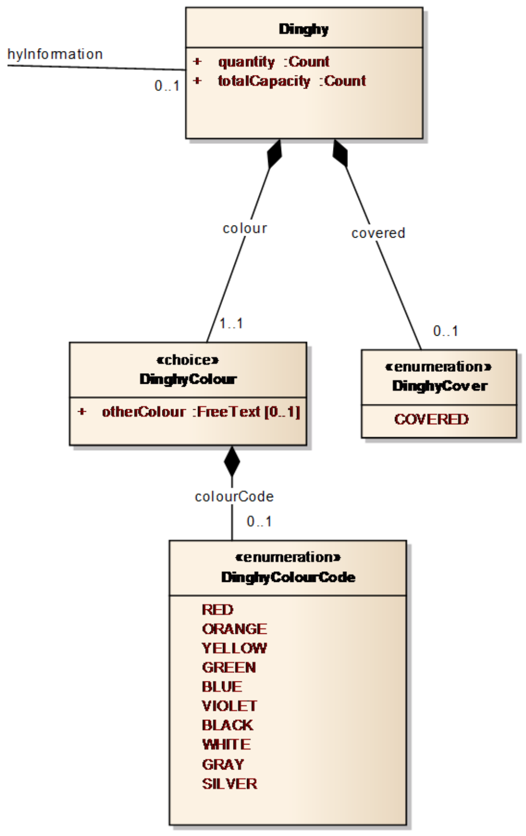


Figure 6 - Use of «choice»Stereotype

Best Practice 47 - Set of stereotypes is restricted

Best Practice 48 - "Normal" Flight Object classes have no stereotype

Best Practice 49 - «datatype» stereotype is used only in Base or Foundation packages

Best Practice 50 - Use «choice» to show alternative properties

Best Practice 51 - Cardinality of choice alternatives is 0..1

3.7 Enumerations

Enumerations present a set of alternative string values used as encodings of some data with a limited set of states, as with "AircraftDangerousGoodsLimitation" enumeration of Figure 2. Some enumerations have only a single defined value, and are used to signal boolean conditions, as discussed in section 3.5.

Since the enumeration values are meant to completely define the states of the enumeration type, they should not normally include values like "OTHER" or "UNKNOWN", unless they reflect legitimate states of the flight data. In cases where the flight data might contain values outside the enumeration values, it is preferred to use a «choice» type containing the enumeration, or an "other" field, as shown in Figure 7.

Best Practice 52 - Enumeration values such as "OTHER" or "UNKNOWN" are discouraged

Best Practice 53 - Use an "otherText" alternative if necessary

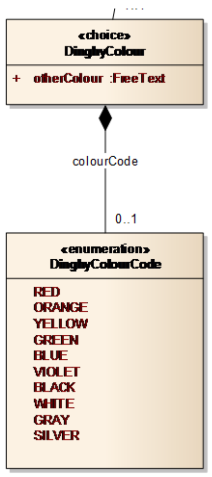


Figure 7 - Representing "other" enumeration values

3.8 Documentation

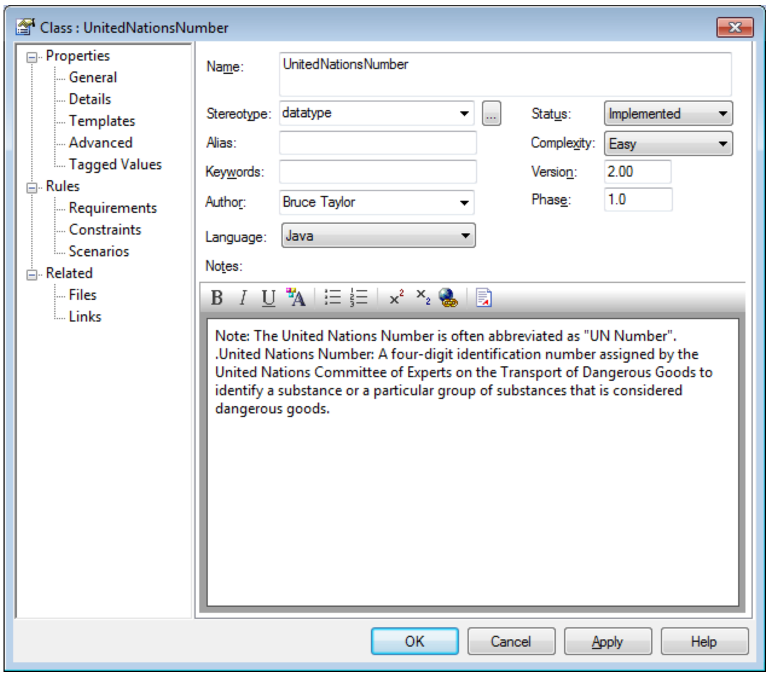
Every component of the FIXM data model: packages, classes, and properties should contain documentation that explains its usage for benefit of data modelers, readers, or programmers who have to work with the model.

Documentation should be applied as near as possible to the location in the model where the data is defined. For example, it is preferred to document a model attribute or relation, rather than its containing class, so in Figure 2 the documentation for “departureCountry” should appear on the attribute, rather than in the documentation for the “ShippingInformation” class.

Documentation of any model element should be able to stand alone, without having to refer to other elements.

The documentation is very often derived directly from the FIXM Data Dictionary, but since some container classes do not have direct relatives in the directory, the data modelers need to supply definitions for these elements. Lines that begin with a period (“.”) are presumed to originate in the data dictionary, and are subject to replacement as data dictionary entries are changed, but lines that begin with any other character are presumed to be supplied by the data modelers and will be protected against automatic erasure or modification. An example of this dichotomy is shown in Figure 8.

When documentation is copied from the FIXM Data Dictionary to the FIXM Logical Model entry, line breaks are preserved but long lines are not artificially broken, so when text wraps in the Enterprise Architect window it looks like multiple lines. In fact, the second through fifth lines of Figure 8 are one long line of text, wrapped by Enterprise Architect.

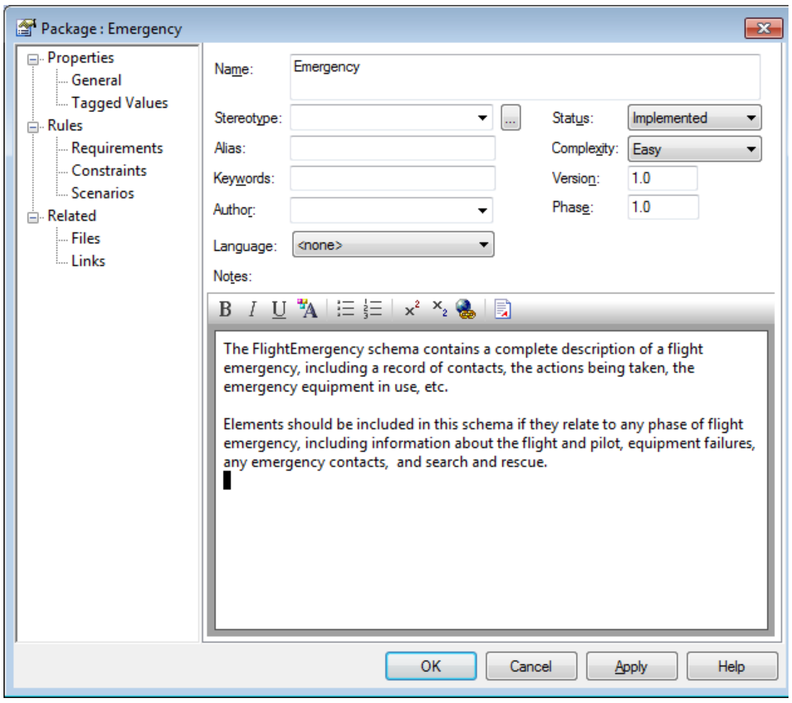


Name: UnitedNationsNumber
Stereotype: datatype
Status: Implemented
Alias:
Complexity: Easy
Keywords:
Version: 2.00
Author: Bruce Taylor
Phase: 1.0
Language: Java
Notes:

Note: The United Nations Number is often abbreviated as "UN Number".
.United Nations Number: A four-digit identification number assigned by the United Nations Committee of Experts on the Transport of Dangerous Goods to identify a substance or a particular group of substances that is considered dangerous goods.

Figure 8 - Element Documentation

The documentation for packages should contain an explanation for the package’s content and usage, including the qualities that unite all the components of the package, the role of the package in the overall FIXM Logical Model, and any special rules or patterns that apply to the package or its contents.



Name: Emergency
Stereotype:
Alias:
Keywords:
Author:
Language: <none>
Status: Implemented
Complexity: Easy
Version: 1.0
Phase: 1.0
Notes:

The FlightEmergency schema contains a complete description of a flight emergency, including a record of contacts, the actions being taken, the emergency equipment in use, etc.

Elements should be included in this schema if they relate to any phase of flight emergency, including information about the flight and pilot, equipment failures, any emergency contacts, and search and rescue.

Figure 9 - Example of package documentation

Best Practice 54 - All model components should be documented

Best Practice 55 – Data dictionary comments are considered adequate documentation

Best Practice 56 - Add extra documentation as needed for clarity

Best Practice 57 - Apply documentation where data is defined

Best Practice 58 - Documentation should not refer to other model objects

Best Practice 59 - Package documentation should describe the theme of the package

3.9 Constraints

For many data items, the FIXM Data Dictionary describes constraints on the object's size, value range, or lexical pattern. These can be captured in the "Constraint" table of the data model, as shown in Table 1 and will be used by the schema generator to produce the appropriate restriction facets in the XSDs. The constraint syntax is chosen to be representation neutral, and the constraint type is taken from Table 1.

Best Practice 60 - Use Constraints to capture limitations on data values

Best Practice 61 - Use Constraints to direct XSD generation

Type	Format	Description
PATTERN	XSD regular expression	A regular expression pattern that defines the allowed lexical structure of the element text. See http://www.w3.org/TR/xmlschema-2/ for definition.
RANGE	[low..high] or (low..high) or [low..high) or (low..high]	Lower and upper bounds on the value range. Brackets ("[]") indicate inclusive containment, parentheses ("()") indicate exclusive containment, and mixed pairs ("[)" or "(]") indicate mixed containment.
FRACTION	[digits,precision]	"Digits" represents the maximum number of digits in the fraction, and "precision" represents the number of digits to right of decimal. For example, a decimal number of the form "1234.56" would have a fraction constraint of "[6,2]".
LENGTH	[low..high]	"Low" is the minimum length of the string, and "high" is the maximum length, both inclusive.
DEFAULT	Value	Specifies the default value for the data. Value must be a legal representation of the data type.
CONSTANT	Value	Specifies the default value for the data, and implies that the value cannot be modified. Value must be a legal representation of the data type.
NILLABLE	True/False	Indicates that the element can be nillable. This is used to guide schema generation to allow nillable elements. Nil value in an element indicates that a value has been cleared from previously set value.

Table 1 - FIXM Constraint Types

3.10 Ordering and Duplication

Many of the FIXM attributes and relationships have a cardinality of 0..* or 1..*, making them collections of entities. The default semantics of these collections is that they are not ordered (that is, entities may appear in arbitrary order) and do not allow duplication of items within the collection. In mathematical terms, the default definition of a collection is a set, rather than a list.

These semantics may be altered by setting the “ordered” and “allow duplicates” properties of the collection using Enterprise Architect. Figure 10 and Figure 11 illustrate how to set these properties in attributes and the target end of relations, respectively.

If the “ordered” checkbox is selected, then the list is presumed to be sorted according to the natural ordering of its contained types. Natural ordering means value comparison for primitive types, but it is not possible to specify the ordering relationship for structured types, except in documentation. It is not possible to specify whether the ordering is ascending or descending. If the “allow duplicates” checkbox is selected, then the list may contain multiple equivalent items, where equivalence means value equality for primitive types. It is not possible to specify the equivalence test for structured entity types except in documentation.

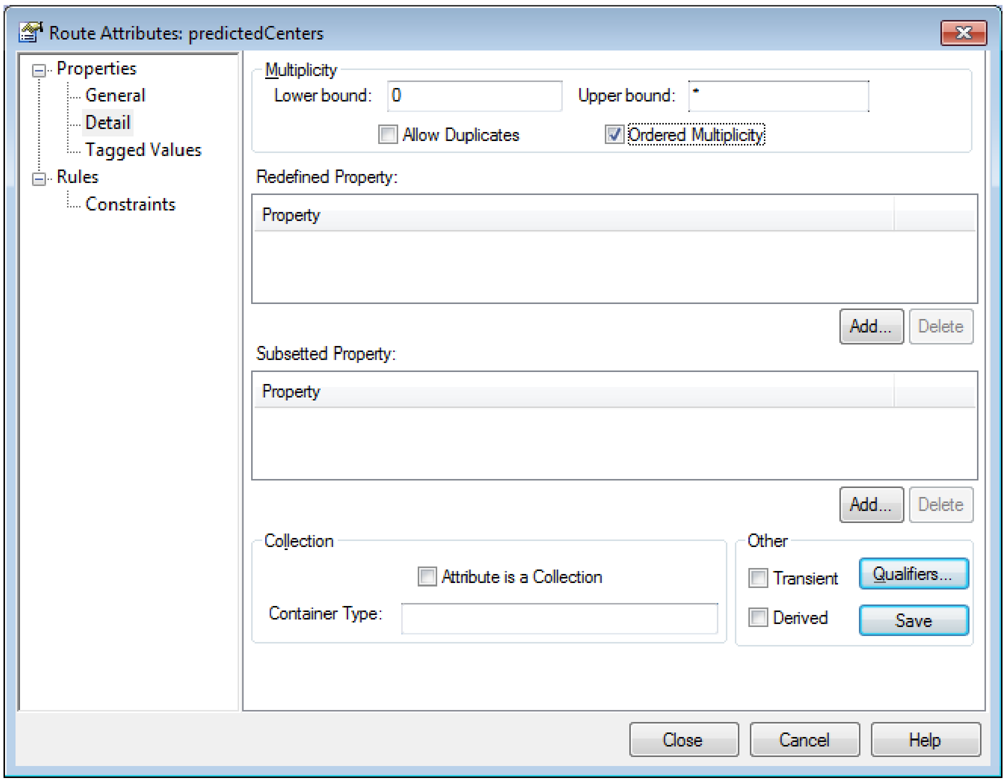
Best Practice 62 - Indicate ordering of attributes and relationships

Best Practice 63 - Indicate when attributes and relationships contain duplicate values

Best Practice 64 - When ordering of structured types is specified, the documentation must make the ordering relation explicit

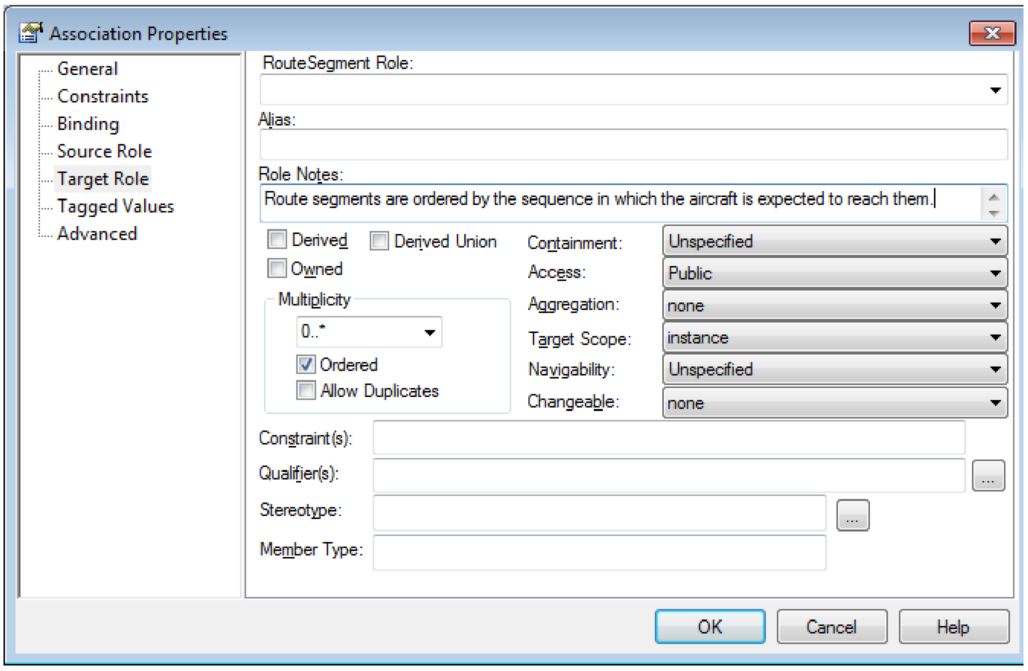
Best Practice 65 - By default an order relation is ascending

Best Practice 66 - If an order is descending, this must be stated explicitly in documentation



Multiplicity
Lower bound: 0
Upper bound: *
[ ] Allow Duplicates
[✓] Ordered Multiplicity
[ ] Attribute is a Collection
Container Type:
[ ] Transient
[ ] Derived

Figure 10 - Specifying Ordering in an Attribute



RouteSegment Role:
Alias:
Role Notes: Route segments are ordered by the sequence in which the aircraft is expected to reach them.
[ ] Derived
[ ] Derived Union
[ ] Owned
Multiplicity: 0..*
[✓] Ordered
[ ] Allow Duplicates
Containment: Unspecified
Access: Public
Aggregation: none
Target Scope: instance
Navigability: Unspecified
Changeable: none
Constraint(s):
Qualifier(s):
Stereotype:
Member Type:

Figure 11 - Specifying Ordering of a Relationship

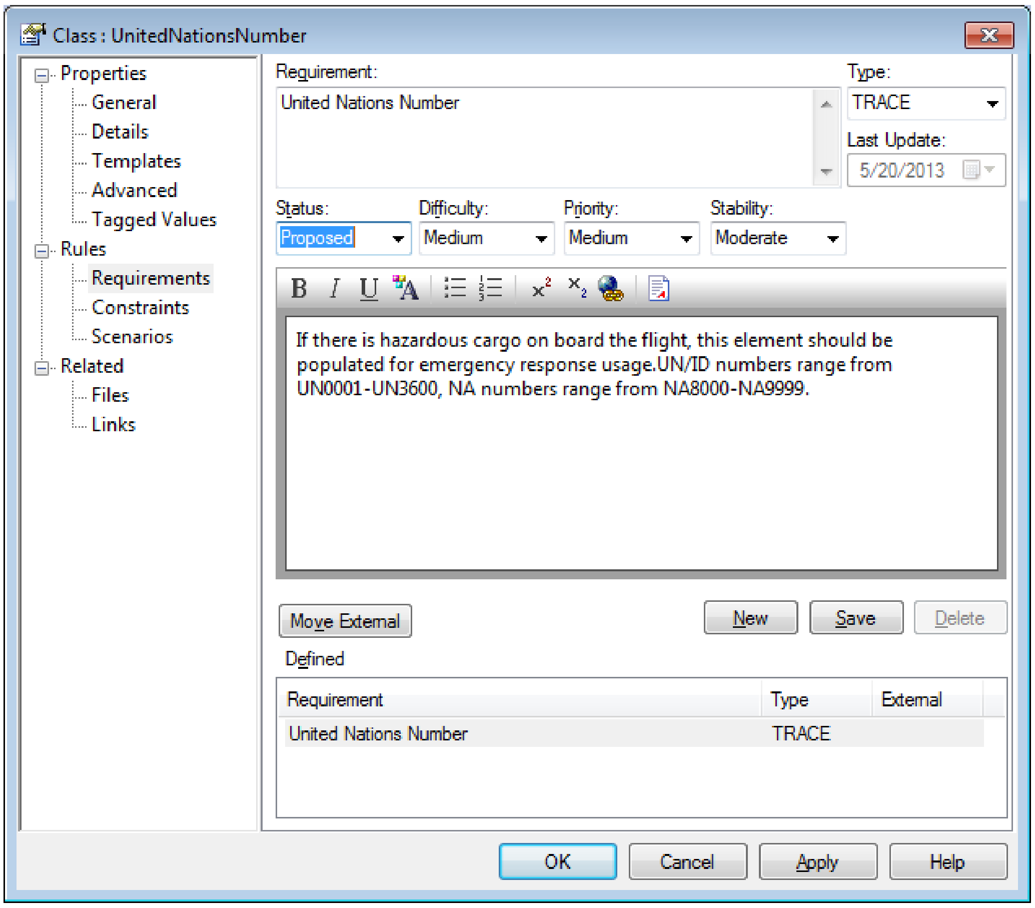
3.11 Requirements

The “Requirements” section of the Enterprise Architect data may contain one or more references back to the FIXM Data Dictionary, as shown in Figure 12. In Enterprise Architect, neither attributes nor relations have storage for requirements, so when Data Dictionary entries are mapped to attributes or relations, their requirements appear in either the source or the target entity, whichever is the closest match to the data dictionary entry.

All requirements are of type “TRACE”, and the Status, Difficulty, Priority, and Stability fields are not used.

Best Practice 67 - Use Requirements of type TRACE to map data dictionary entries

Best Practice 68 - Relations and attributes map to their containing classes



Requirement: United Nations Number

Type: TRACE

Last Update: 5/20/2013

Status: Proposed | Difficulty: Medium | Priority: Medium | Stability: Moderate

If there is hazardous cargo on board the flight, this element should be populated for emergency response usage.UN/ID numbers range from UN0001-UN3600, NA numbers range from NA8000-NA9999.

Defined

Requirement	Type	External
United Nations Number	TRACE	

Figure 12 - Requirements and Files

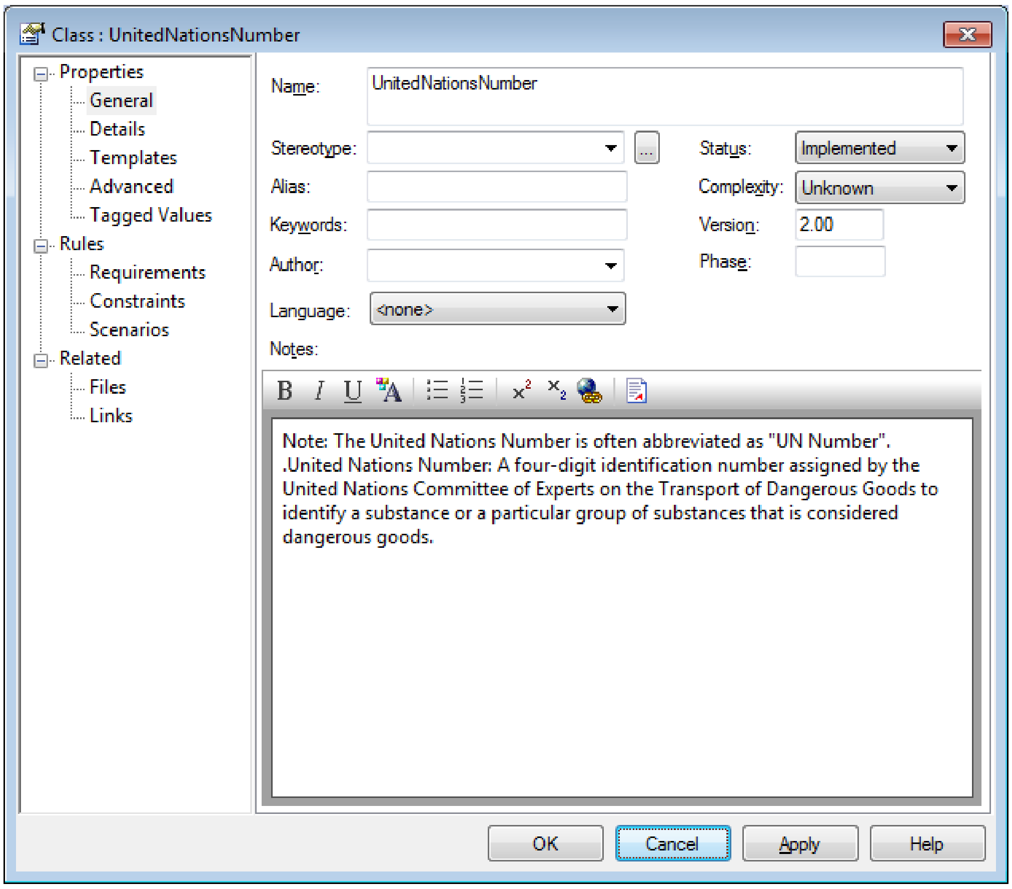
3.12 Version

Every data model entity is marked with "Version" metadata element that indicates the version of the FIXM Logical Model when the element was created. The element version may be set manually or automatically, and appears in the "General" display for the entity, as shown in Figure 13. The Phase field is not used.

The version for an attribute is the version of its containing entity, and the version for a relation is the more recent version of its source or target entity type.

Best Practice 69 - Version marks when model element was created

Best Practice 70 - Version of attributes and relations map to containing type



Name: UnitedNationsNumber

Stereotype:

Alias:

Keywords:

Author:

Language: <none>

Status: Implemented

Complexity: Unknown

Version: 2.00

Phase:

Notes:

Note: The United Nations Number is often abbreviated as "UN Number".
.United Nations Number: A four-digit identification number assigned by the United Nations Committee of Experts on the Transport of Dangerous Goods to identify a substance or a particular group of substances that is considered dangerous goods.

Figure 13 - Element Status, Version, Authorship

3.13 Authorship

Elements of FIXM are presumed to be "authored" by the entire data modeling team and not by an individual, so the "Author" field should be blank as shown in Figure 13. The Enterprise Architect tool will enter the user's login name into the "Author" field when the entity is created, but this text should be deleted, either manually or automatically.

Best Practice 71 - Model elements should not be related to individual authors

3.14 Status

Every data model element is marked with a “Status” metadata that indicates the current state of the element, shown in Table 2. The status of the elements may be updated manually or automatically, and appears in the General window for the element, as shown in Figure 13.

Best Practice 72 - Use status to indicate element's life cycle stage

Best Practice 73 - Status of Attributes and Relations map to containing types

Status Name	Definition
Proposed	The element has identified as a candidate for implementation. Elements with status “Proposed” are likely to be place holders for further implementation.
Implemented	The element has been implemented by the data modeling team, but not yet tested or accepted.
Validated	The element has passed all validation tests.
Approved	The element has been accepted by the Change Control Board (CCB). Further changes to the element require approval from the CCB.

Table 2 - Life Cycle Statuses

3.15 Local Data Types

When defining local data types, it is important to consider whether information can be expressed using an existing type or whether a new type definition is needed. It is recommended to reuse an existing type if it contains sufficient information.

But, as with all best practices, there are allowable exceptions to this rule. In Figure 14 the type StandardInstrumentRouteDesignator contradict this best practice because it adds neither information nor pattern to the string type it extends, however since it represents an important concept and key element of the Data Dictionary, it seems worthwhile to represent it as its own class to emphasize this importance. It is also used by several other classes within the model. Choosing to model an entity as an attribute or a class will always be a judgement call on the part of the modeller and this best practice is a recommendation rather than a prohibition.

Best Practice 74 - Avoid empty extensions where they add no information

Best Practice 75 - Use empty extensions if they clarify intent

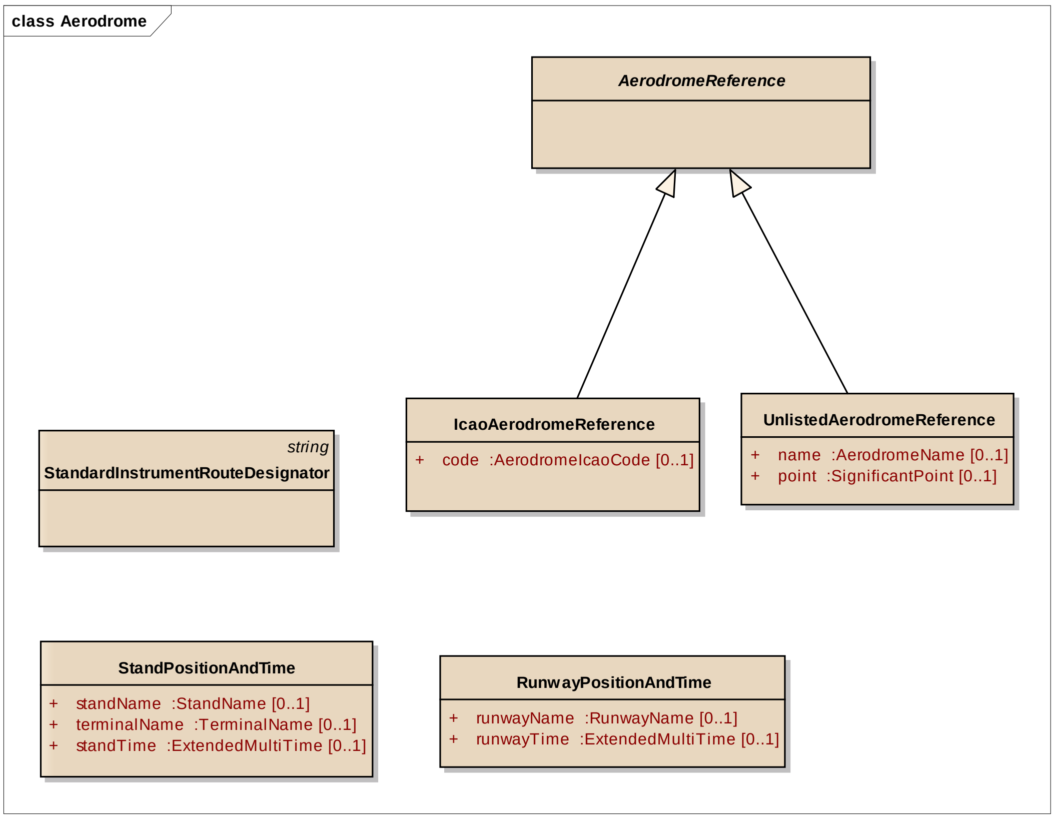

class Aerodrome
AerodromeReference
string
StandardInstrumentRouteDesignator
IcaoAerodromeReference
+ code :AerodromeIcaoCode [0..1]
UnlistedAerodromeReference
+ name :AerodromeName [0..1]
+ point :SignificantPoint [0..1]
StandPositionAndTime
+ standName :StandName [0..1]
+ terminalName :TerminalName [0..1]
+ standTime :ExtendedMultiTime [0..1]
RunwayPositionAndTime
+ runwayName :RunwayName [0..1]
+ runwayTime :ExtendedMultiTime [0..1]

Figure 14 - Empty Extension Emphasizes Important Concept

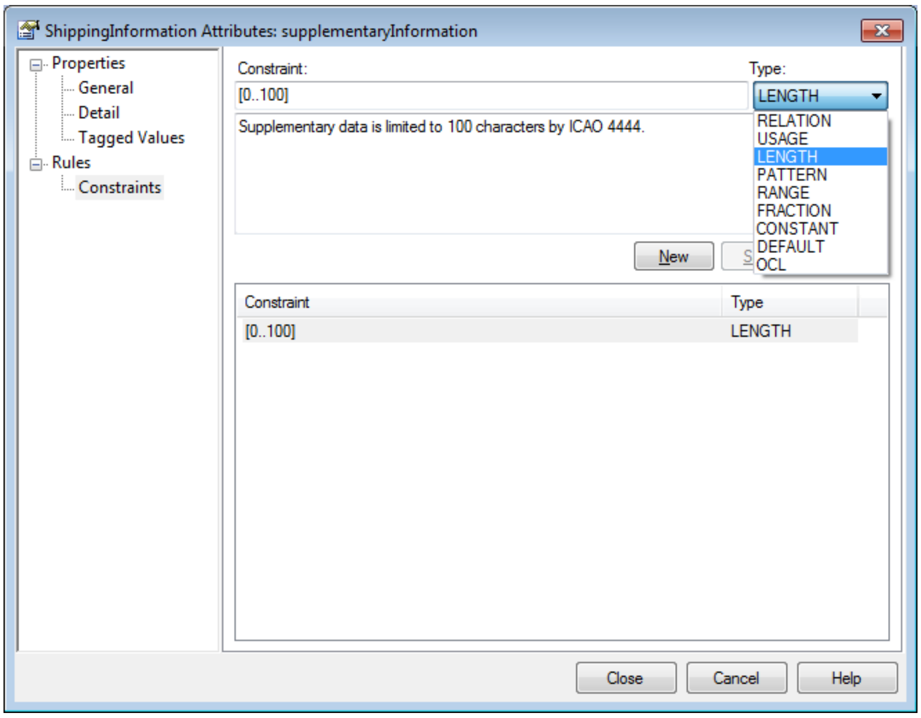
3.16 Constraining Property Values

Most of the data entries of the FIXM Data Dictionary contain some description of constraints on the "Basic" data type, and it is part of the data modeling process to capture those constraints in the data model. The set of available constraints is shown in Table 1, and they can be applied to the three main UML elements in the diagram:

- Classes
  Class constraints implicitly restrict the value of all properties of the class type, and any classes derived from the class, so they are suitable for defining basic types and types that are expected to be reused in multiple contexts.
- Attributes
  Attribute constraints restrict the value of the immediate data value, with no effect on other instances. They are best used to restrict value of class properties.
- Relationships
  Relationship constraints act just like attribute constraints, but they are applied to the target type of the relationship. They are best used when a class is referenced multiple times, each reference having its own restrictions.

All these kind of constraints are set in the same way, by editing the "Constraint" table of the class, attribute, or relationship using Enterprise Architect. Figure 15 illustrates the steps in constraining "ShippingInformation.supplementalData" to a length of 100 characters or fewer. Setting the other constraint types, and setting constraints for Classes and Relationships follows the same model.

Best Practice 76 - Use Constraints to capture data restrictions from the data dictionary


ShippingInformation Attributes: supplementaryInformation

Constraint: [0..100]

Type: LENGTH

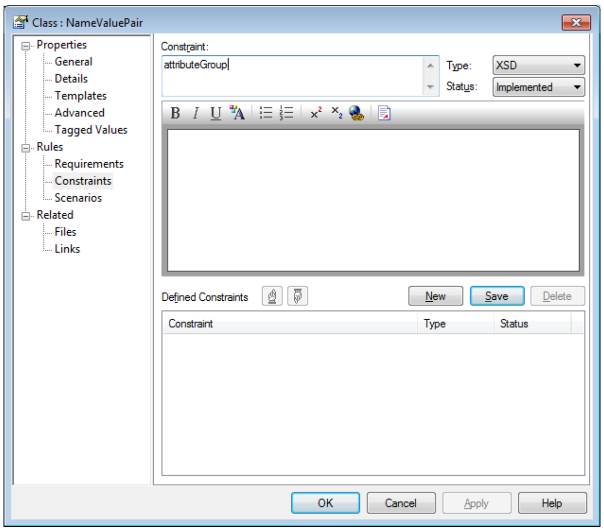
Supplementary data is limited to 100 characters by ICAO 4444.

Constraint	Type
[0..100]	LENGTH

Figure 15 - Constraining an Attribute's Length

3.17 Guiding XSD Generation

In FIXM the schema XSD files are produced from the logical model by a schema generation tool in the FIXM Workbench. For the most part, schema generation is automatic; the tool recognizes model patterns and generates the appropriate XSD content. But in some cases, the tool does not have enough context to determine the optimal XSD content, and the modeller needs to supply “hints” to the tool to achieve the best possible schemas. These hints are supplied using the XSD “constraint” type, as shown in Figure 16.



Constraint: attributeGroup

Type: XSD

Status: Implemented

Figure 16 - Specifying an XSD Constraint

The types of XSD “constraints” are shown in Table 3. Note that the constraint values are case insensitive.

Table 3 – XSD Constraints

Constraint	Applies To	Effect
Attribute	Properties	Forces generation as an XSD <attribute>. Only properties that reference simple types and primitive types can be generated as attributes. Attributes produce significantly smaller XML footprints than do elements.
Element	Properties	Forces generation as an XSD <element>. Sometimes simple type references should be generated as elements to keep them together with related elements.
AttributeGroup	Classes	Forces generation as an XSD <attributeGroup>. All the contents of the class (which must be simple types or primitive types) are produced as attributes, and any reference to the attribute group includes those attributes into the referring type. For frequently used collections of simple types, this can often dramatically reduce generated XML.
Optional	Properties	For any property generated as an XSD <element> appends the qualifier nillable=’true’. (Note: this is the default setting for element generation.)
Required	Properties	For any property generated as an XSD <element> appends the qualifier nillable=’false’.
Inline	Properties	Copies the contents of the referenced class into the referencing type instead of generating a type reference (“inlining”). This avoids one level of scope tags in the generated XML, and can be

		a useful optimization for frequently used types.

3.18 Features

The concept of a "feature" is derived from the GML modeling standard, where it captures information about a single geographical elements: lake, building, mountain, etc. In FIXM, a "feature" encapsulates the information about a single element of the flight: its route, its arrival, or the aircraft involved. Data belongs in a feature if they share the following characteristics:

1. They are gathered from the same source: radar, flight information system, pilot, controller, etc.
2. They are gathered at approximately the same moment and represent the state of the flight at that moment.
3. They should logically be created, updated, and deleted together as an atomic unit.

A notable exception to these rules is the Flight class, which is at the centre of FIXM. Although it consists of multiple components from multiple sources gathered at different points throughout the flight cycle, is important to express it as a feature because this will enable to include provenance data pertaining to the entire flight.

Evidently, deciding which data belong to which features requires a deep knowledge of the logical field of Air Traffic Management (ATM) and should be given sufficient thought and review.

As of this writing, the feature decomposition of the FIXM Logical Model is incomplete, and only the highest level elements of the flight are features.

You can create a feature by extending the Feature abstract class. By this extension, your new feature will acquire an optional piece of metadata called the "Provenance." The provenance records the origin of the information in the feature, including:

1. The system that created it.
2. The Air Traffic Control (ATC) unit that created it.
3. Other information about the creator.
4. Time stamp of when the information was created.

It is expected that the Provenance will be created when the feature is first created, then updated whenever any of the data inside the feature is updated.

4 Modeling Extensions

4.1 The Role of Extensions

The FIXM extension mechanism is an explicit recognition that, while the FIXM core models define the internationally shared characteristics of a flight, there will always be a need for region-specific information to accommodate ATM procedures of individual countries and regions. For guidance on placing a data element into the core or in an extension please refer to the FIXM Strategy document.

The FIXM core packages will be developed and maintained by the FIXM development groups, under the aegis of the FIXM CCB and its sponsoring organizations. By contrast, extension models may be developed by any organization, as needed to provide additional flight data. From time to time, elements from extensions may be migrated into the FIXM core packages, under guidance of the FIXM CCB. The overall strategy for managing extensions is described in more detail in the “FIXM Strategy” document. Additionally, the CCB’s process for migrating extensions into the core is described in more detail in the [reference: CCB charter].

The rest of this section describes the process and best practices that will lead to successful extension development, and successful integration with the FIXM core packages.

4.2 Extension Best Practices

4.2.1 General Practices

All best practices described in sections 1 through 3 of this document should be followed in an extension model.

4.2.2 Extension Isolation

It is intended that every extension be able to stand by itself, without reference to other extensions. So, extension models may refer to classes defined within their own packages, to classes defined in the FIXM Core packages, or to classes defined in the FIXM base and foundation classes. Inter-extension references are strongly discouraged, because they lead to version dependencies between the extensions, and consequent difficulty in building and validating XML messages that contain extension data.

Best Practice 77 - Extensions may refer to their own data elements.

Best Practice 78 – Extensions may refer to data elements in the FIXM core packages.

Best Practice 79 – Extensions should not refer to data elements defined in other extensions.

4.2.3 Name Qualification

The names of each class within an extension should carry a short prefix that distinguishes it from elements of the core packages, when the names may be confused. Figure 17 illustrates the use of the prefix “Nas” to distinguish the FAA National Airspace System (NAS) extension arrival class from the core FlightArrival class, because the similarity of names is likely to lead to confusion. However, the “LandingLimits” class defines a concept that exists only within the NAS extension, so the prefix is omitted.

Best Practice 80 - Use a short prefix to distinguish extension from core classes.

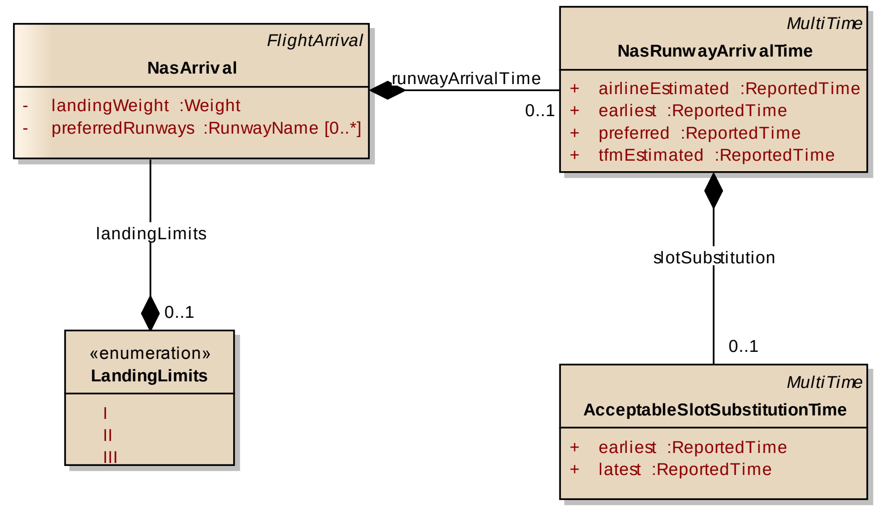


Figure 17 - Use of prefixes to qualify names

4.2.4 Extension Strategy 1: Use of UML Inheritance

All classes in an extension fall into one of two categories:

1. They are unique to that extension and have no corresponding class in the core packages, or
2. They are an extension of some class in the core package, adding data elements or redefining data elements in some way.

For case (1), when classes are unique to the extension, no special techniques are necessary: the classes can simply be defined according to normal best practices liked to the rest of the model. But for case (2), when classes extend or modify objects from the core packages, the best practice is to use UML inheritance to define an extension type containing the extra or modified data elements.

Figure 18 illustrates this pattern: the NAS extension requires the definition of an alternative aerodrome reference with a name that matches the International Air Transport Association (IATA) aerodrome pattern, to use in place of the core aerodrome reference that uses the International Civil Aviation Organization (ICAO) pattern. In this figure, we see that the “NasAerodromeReference” class is defined as an extension of the “IcaoAerodromeReference” class from the core packages, adding its own “nasCode” aerodrome name property. In constructing a NAS extension message, the “NasAerodromeReference” class can be used anywhere that the “IcaoAerodromeReference” class is valid. Applications that expect the NAS extensions will use either the ICAO or IATA Aerodrome fields, but applications that are not expecting a NAS extension will use only the ICAO field.

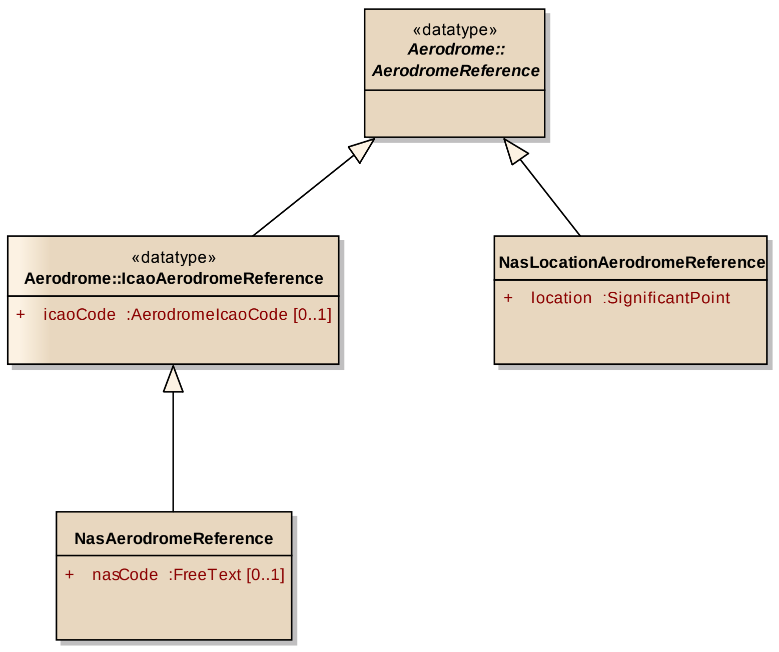

«datatype»
Aerodrome::
AerodromeReference
«datatype»
Aerodrome::IcaoAerodromeReference
+ icaoCode :AerodromeIcaoCode [0..1]
NasLocationAerodromeReference
+ location :SignificantPoint
NasAerodromeReference
+ nasCode :FreeText [0..1]

Figure 18 – Extending a base class to change constraints

Best Practice 81 – Add data to core classes using UML inheritance.

Of course, this pattern can be applied to other, higher level classes. Figure 19 illustrates how to extend the core Flight class to implement the extension NAS flight class, which would appear in NAS specific messages in place of the FIXM core class.

Best Practice 82 - Extend FIXM Core to define new flight data

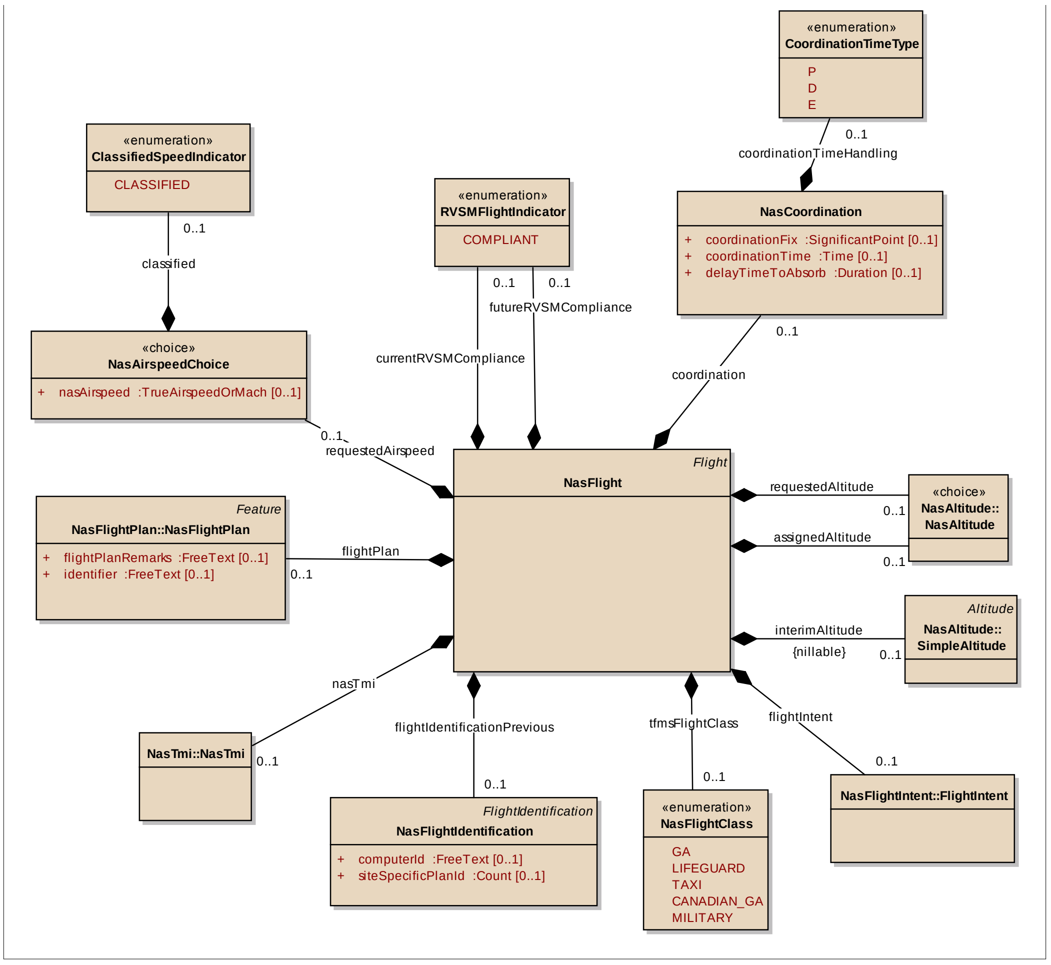


Figure 19 - Extending a Core Flight

4.2.5 Constraint Redefinition Forbidden

A common pattern in extension development is the need to change or extend the restrictions on a core type. For example, the NAS extension uses IATA aerodrome codes (3-4 characters including digits) instead of ICAO codes (4 alphabetic characters). It is tempting to simply define an extension type that extends the core type, and then supply a new set of constraints, but this approach is forbidden because different XML validators treat this overriding in different ways, and the result of validating such an extension is undefined.

Instead, follow the pattern shown in Figure 18: the “NasAerodromeReference” class extends the core “IcaoAerodromeReference” class, but adds its own unique code attribute, with a pattern constraint of “[A-Z0-9]{3,4}”. Because “NasAerodromeReference” is an extension, it can be used in any place that an “IcaoAerodromeReference” is valid, but its data field will not be confused with the field of the core class.

Best Practice 83 - Constraint redefinition forbidden.

4.2.6 Extension Strategy 2: Use of the Extension Class

In some situations, it is advisable to keep the extension data completely segregated within the overall set of flight models, either to keep it separate from the core flight data, or to separate it from other extension data. In this situation, the strategy of defining extension types is not appropriate. Instead, make use of the Extension type from the core packages to define a separate container for the extension data as shown in Figure 20.

Recall that the core Flight class contains an optional set of Extension classes. If an extension defines a class that derives from Extension, then the core Flight class can carry one or more instances of your extension container. Applications that need the extension data can locate the proper Extension subtype in the list, while applications that use only the core classes can simply ignore the extensions.

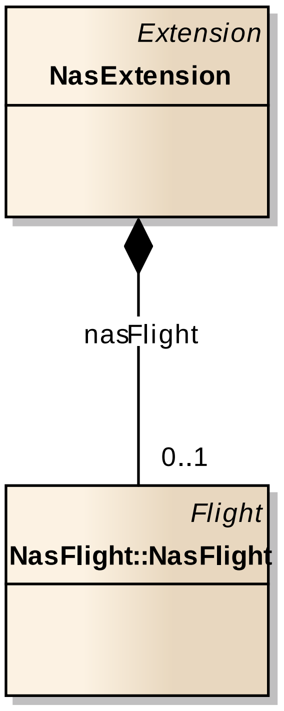


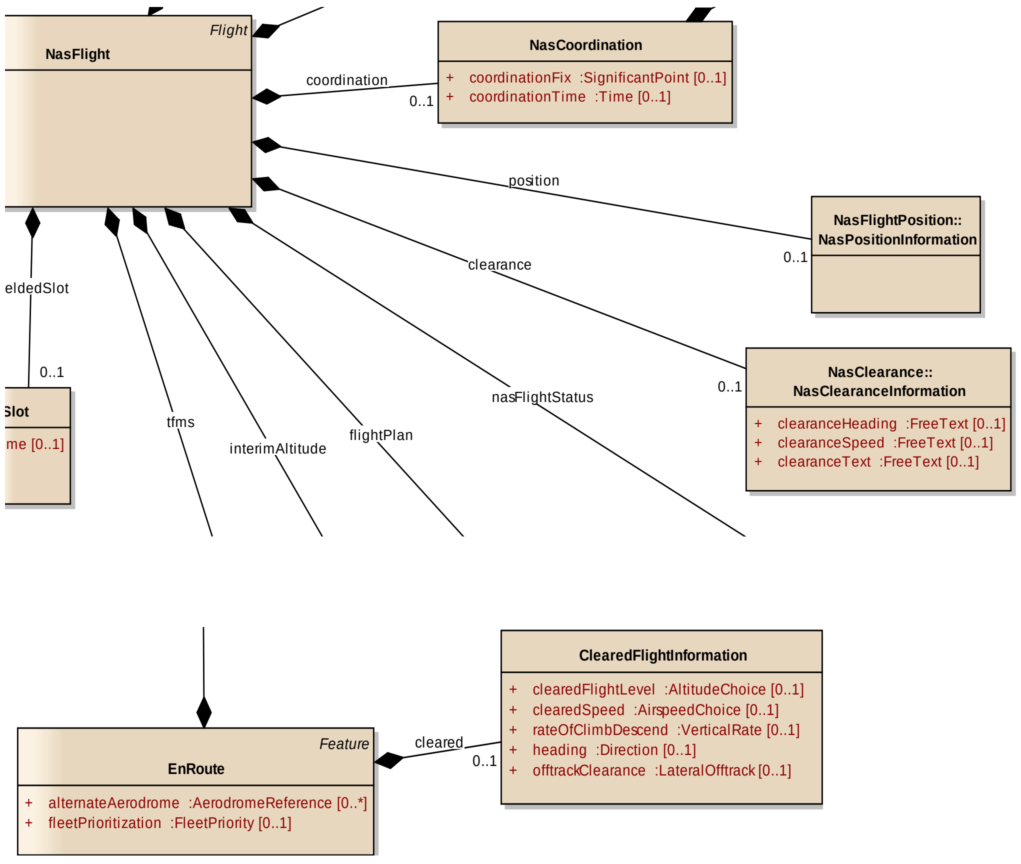
Figure 20 - Defining an Extension Subclass

4.2.7 Comparison of Strategies

It is important to compare the two aforementioned strategies for defining extensions. Strategy 1 is the most natural approach and analogous to the software engineering practice of class inheritance. There is no ambiguity where the data should be placed. Extended classes may replace the core equivalents at any location in the model where the core version appears.

Strategy 2, while allowing the convenience of grouping the extended classes in a separate area from the core, may present some significant confusion and potential of data duplication.

For instance let's consider a scenario presented in Figure 21. The element NasClearanceInformation contains additional clearance information that supplements the core version. It is directly associated with the NASFlight element whereas in the core, the clearance information is associated with the EnRoute class. Therefore there are multiple paths to similar information in different areas of the model. Furthermore, since the NasFlight is attached to the core Flight through the extension association, and itself inherits form the core Flight class, it is possible to create an instance of Flight containing certain fields, and define the same fields in the NasFlight since those fields are available e by inheritance.



Flight
NasFlight
coordination
0..1
NasCoordination
+ coordinationFix :SignificantPoint [0..1]
+ coordinationTime :Time [0..1]
position
NasFlightPosition::
NasPositionInformation
0..1
clearance
eldedSlot
0..1
NasClearance::
NasClearanceInformation
0..1
+ clearanceHeading :FreeText [0..1]
+ clearanceSpeed :FreeText [0..1]
+ clearanceText :FreeText [0..1]
Slot
me [0..1]
tfms
interimAltitude
flightPlan
nasFlightStatus
ClearedFlightInformation
+ clearedFlightLevel :AltitudeChoice [0..1]
+ clearedSpeed :AirspeedChoice [0..1]
+ rateOfClimbDescend :VerticalRate [0..1]
+ heading :Direction [0..1]
+ offtrackClearance :LateralOfftrack [0..1]
Feature
EnRoute
+ alternateAerodrome :AerodromeReference [0..*]
+ fleetPrioritization :FleetPriority [0..1]
cleared
0..1

Figure 21 – Inconsistency between placement of extended vs. core class

A recommended approach would be to define a NasClearanceInformatoin class that inherits from core ClearedFlightInformation as presented inFigure 22. This would enable to place the extended class where the core ClearedFlightInformation appears, thus replacing it. Also, to avoid data duplication, NasFlight class should replace the core Flight, rather than attach to it using the Extension attribute.

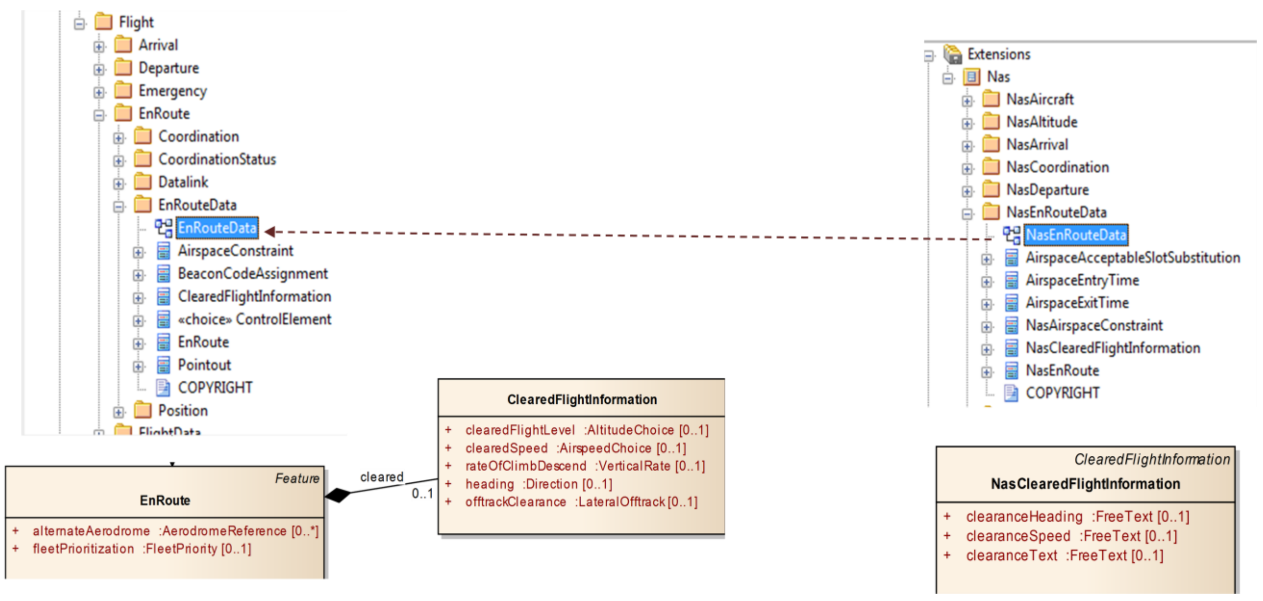

Flight
Arrival
Departure
Emergency
EnRoute
Coordination
CoordinationStatus
Datalink
EnRouteData
EnRouteData
AirspaceConstraint
BeaconCodeAssignment
ClearedFlightInformation
«choice» ControlElement
EnRoute
Pointout
COPYRIGHT
Position
Extensions
Nas
NasAircraft
NasAltitude
NasArrival
NasCoordination
NasDeparture
NasEnRouteData
NasEnRouteData
AirspaceAcceptableSlotSubstitution
AirspaceEntryTime
AirspaceExitTime
NasAirspaceConstraint
NasClearedFlightInformation
NasEnRoute
COPYRIGHT
ClearedFlightInformation
+ clearedFlightLevel :AltitudeChoice [0..1]
+ clearedSpeed :AirspeedChoice [0..1]
+ rateOfClimbDescend :VerticalRate [0..1]
+ heading :Direction [0..1]
+ offtrackClearance :LateralOfftrack [0..1]
Feature
EnRoute
+ alternateAerodrome :AerodromeReference [0..*]
+ fleetPrioritization :FleetPriority [0..1]
cleared
0..1
ClearedFlightInformation
NasClearedFlightInformation
+ clearanceHeading :FreeText [0..1]
+ clearanceSpeed :FreeText [0..1]
+ clearanceText :FreeText [0..1]

Figure 22– Recommended approach for placing extended class

Best Practice 84 - Use UML Inheritance strategy to define extensions

4.3 Extension Projects

Developing a FIXM extension requires that it be modelled using Enterprise Architect, the standard UML modeling tool for FIXM. Figure 23 illustrates the required structure of an Enterprise Architect extension model. The extension model must contain the core packages, in addition to the extension packages, because Enterprise Architect requires that all references be resolved within a single project. The most straightforward approach to producing an extension model is:

1. Obtain a copy of the most recently released core model, named FIXM.eap.
2. Rename FIXM.eap to <your extension>.eap
3. Use Enterprise architect to create a new Model node named “Extensions”
4. Under the Extensions model, create a top level package for your extension.
5. Under your top level package, create any needed sub-packages.

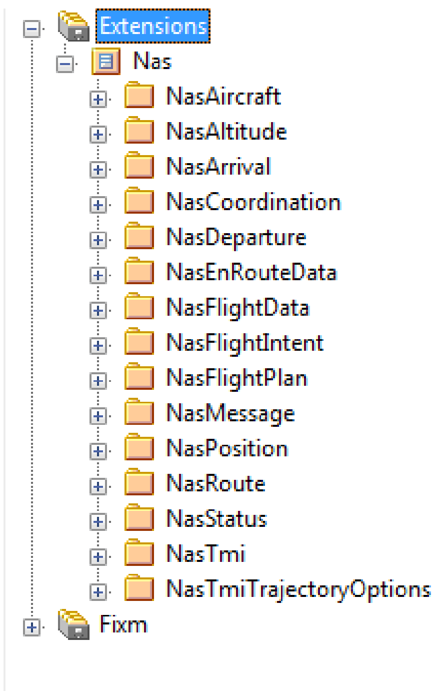

Extensions
Nas
NasAircraft
NasAltitude
NasArrival
NasCoordination
NasDeparture
NasEnRouteData
NasFlightData
NasFlightIntent
NasFlightPlan
NasMessage
NasPosition
NasRoute
NasStatus
NasTmi
NasTmiTrajectoryOptions
Fixm

Figure 23 - Structure of an Enterprise Architect Extension Project

Appendix A FIXM Copyright Notice

Best Practices Summary