Flight Information Exchange Model

Modelling 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.

December 15, 2017

Version: 4.1.0

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
4.0.0	31/10/2016	MIT LINCOLN LABORATORY	FIXM v4.0.0 Release
4.1.0	15/12/2017	MIT LINCOLN LABORATORY	FIXM v4.1.0 Release

Table of Contents

Table of Figures

Table of Tables

References

1. FIXM Operational Data Description
2. FIXM Change Management Charter, v1.1, FIXM CCB
3. FIXM Strategy, v1.1, FIXM CCB
4. FIXM Development Guidelines, v2, FIXM CCB

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 Operational Data Description (FIXM ODD) and implementation artefacts 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 Modelling Language) class diagram. The UML class diagram is a schema-neutral format that can be understood and reviewed by people with various backgrounds. 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 Operational Data Description

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

1. Items derived from the FIXM ODD: This class of item is a direct mapping from one or more entries and contains information about a specific aviation (or related) domain concept. Examples are: Communication Capabilities and Last Contact Radio Frequency.
2. Structural objects: This class of item is used to organize derived items, but (usually) has no direct correspondence in the operational data description. 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 ODD, but have no direct correspondence to the FIXM ODD. Examples are: TextName, PositionPoint, 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 Artefacts

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

1. XSD Schemas
2. Graphical XSD Schema Documentation derived from the XSD Schemas (Using Oxygen[1] or similar XSD tool)

In later releases, the FIXM Logical Model may be used to derive other artefacts such as data access objects, documentation, or simulations. This implies that FIXM should remain implementation neutral, with regards to the physical modelling 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 artefacts 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.

1.4 Data Modelling Tools

1.4.1 Enterprise Architect

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

Best Practice 2 - Enterprise Architect is the primary modelling tool

1.4.2 FIXM Data Modelling Workbench

Other actions required in development of the logical and physical models are provided by the FIXM Data Modelling 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 Modelling Elements

2.1 Packages

Packages are a logical subdivision of the FIXM Logical Model that allow readers and modellers 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, modellers 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 modelling tool.

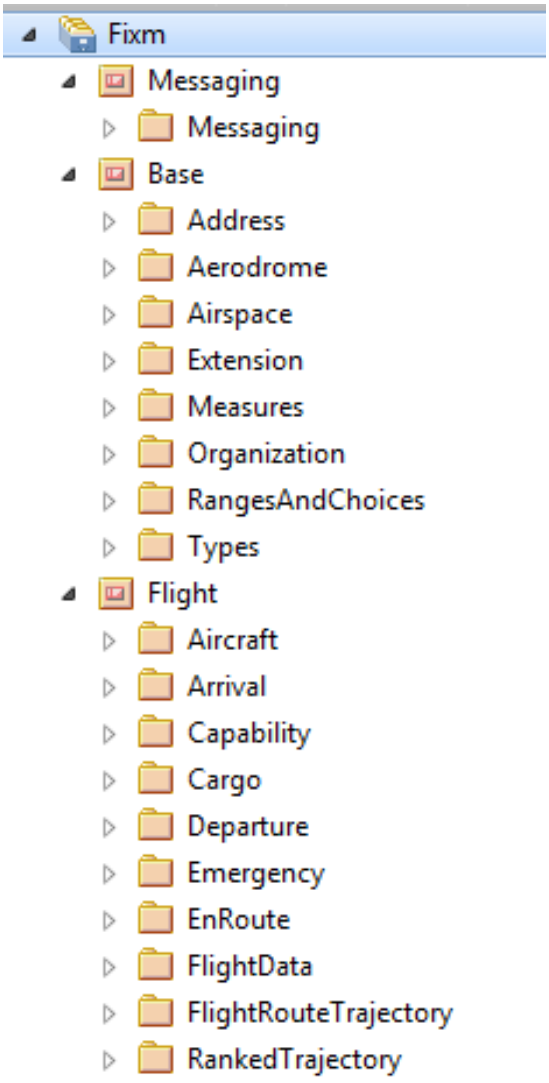


Figure 1 – FIXM Core Package Structure

[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 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. In general, the other packages will refer to and extend classes from the Base packages. Some of the elements contained in the Base packages are modelled after the equivalent versions from the AIXM and GML models.

2.1.2 Flight Packages

The package 'Flight' and its sub-packages contain the elements that derive from the FIXM Operational Data Description, plus structural elements needed to organize those elements.

Best Practice 3 - Limit packages contents to manageable size
Best Practice 4 - All packages have a unifying theme stated in documentation
Best Practice 5 - Commonly shared low level types appear in the Base package
Best Practice 6 - Entities derived from the Operational Data Description appear in the Flight 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 7 - Limit diagram size to two pages Best Practice 8 - Give diagram the same name as its package

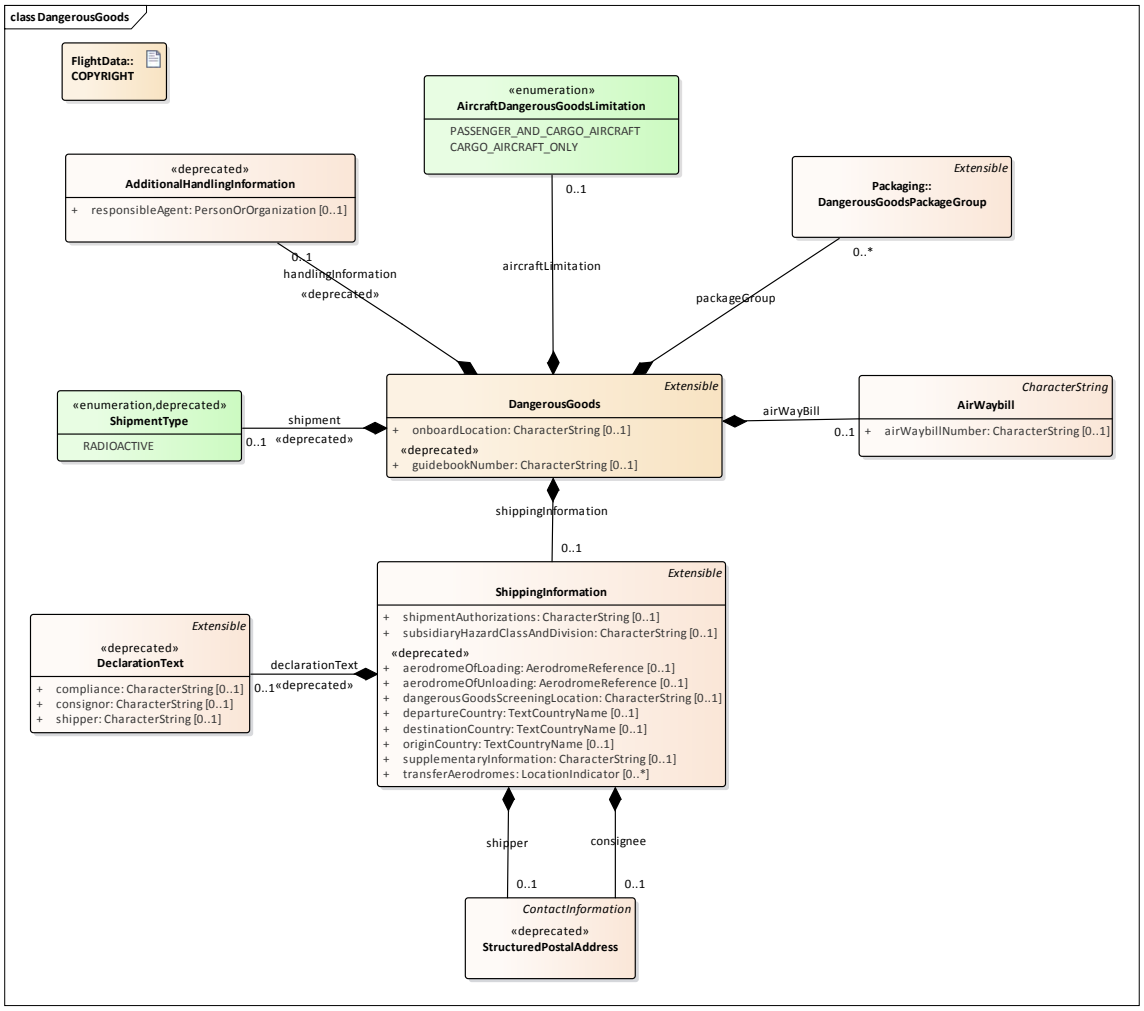


Figure 2 - Example FIXM Diagram

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

Examples of classes from Figure 2 include “ShippingInformation”, “AirWaybill”, and “DeclarationText”.

2.2.3 UML Attributes

A UML "attribute" is a name and datatype pair written inside a class rectangle, as shown by "onboardLocation" 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 0..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 packages. UML attributes are primarily used to avoid cluttering the diagram with references to base types, and obscuring the references among Flight data types.

Best Practice 9 - Attribute datatypes are primitive, or from Base packages
Best Practice 10 - Primitive attribute types allowed only in Base packages
Best Practice 11 - Default attribute multiplicity is 0..1
Best Practice 12 - 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 Flight package. Relations are never used to show a relationship to a class defined in the Base package. This convention is intended to minimize cluttering the diagram and obscuring references among Flight data types.

[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.

Best Practice 13 - All relations are containment composition

Best Practice 14 - Relation connectors do not show directionality arrowhead

Best Practice 15 - Relation names are attached to the connector

Best Practice 16 - Multiplicity is attached to target end of connector

Best Practice 17 - 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 18 - Show both source and target of cross-package references

Best Practice 19 - 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 should be included as a link to a copyright artefact as shown in Figure 2.

Best Practice 20 - 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: EnRoute, 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, aircraftLimitation, etc.
- Enumeration values
  Enumeration values are written all in upper case, using only letters, digits, and the underscore character, as shown in "AircraftDangerousGoodsLimitation " of Figure 2.

Names in the FIXM data model are often taken directly from corresponding entries in the FIXM Operational Data Description, but this correspondence is not required. In general, modellers 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 21 - Name characters limited to upper and lower case, digits and underscore
Best Practice 22 - Use InterCap notation for all names
Best Practice 23 - Use starting capital for packages and classes
Best Practice 24 - Use starting miniscule for attributes and relations
Best Practice 25 - Use all capitals for enumeration values
Best Practice 26 - Names should be expressive of data content or relationship
Best Practice 27 - 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. Choose the British (that is, from the Oxford English Dictionary) when there are alternative English spellings.
4. All names must be unique within their scope (even though Enterprise Architect allows duplicate names).
5. Use singular nouns unless describing an explicit list structure.
6. Use present tense of verbs unless the concept requires past or future.

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

Best Practice 28 - Avoid acronyms unless industry standard
Best Practice 29 - Avoid abbreviations except for very long names
Best Practice 30 - Choose industry standard words and phrases
Best Practice 31 - Choose British spelling when there are alternatives
Best Practice 32 - Names unique within scope
Best Practice 33 - Use singular nouns except for explicit lists
Best Practice 34 - Prefer present tense of verbs
Best Practice 35 - 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 Operational Data Description, and to assist in name matching when mapping operational data description entries to their implementations in the FIXM Logical Model.

Best Practice 36 - Use aliases to capture additional Operational Data Description mapping

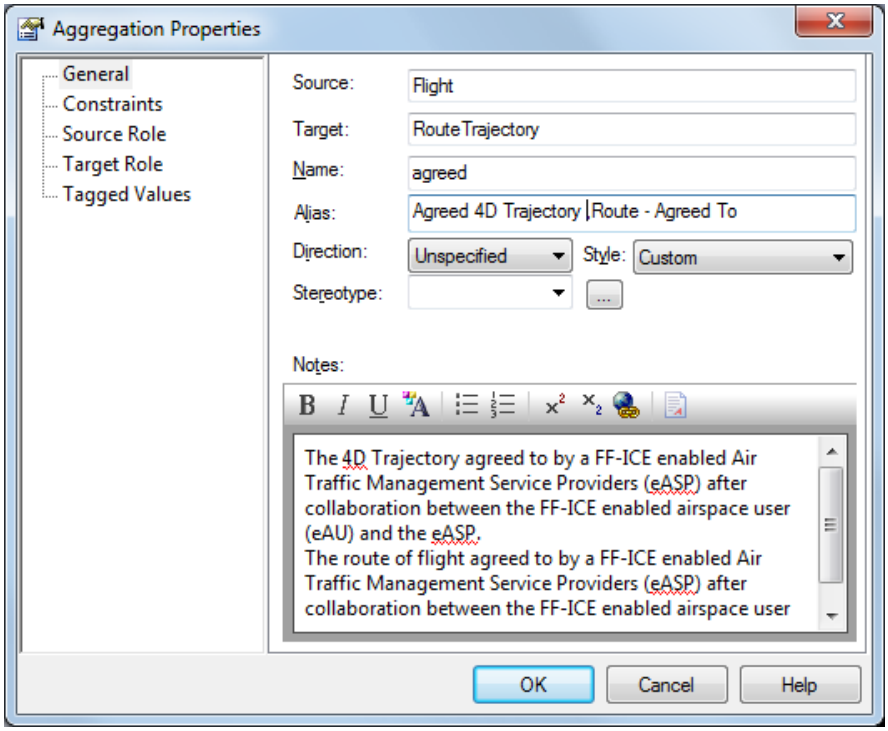


Figure 3 - Use of Alias Names

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 as demonstrated by classes that extend SignificantPoint or by showing the parent class type in italics in the upper right corner as in the class TrajectoryPoint4D. 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 (*SignificantPoint* in Figure 4), and the latter is used when the parent is from another package.

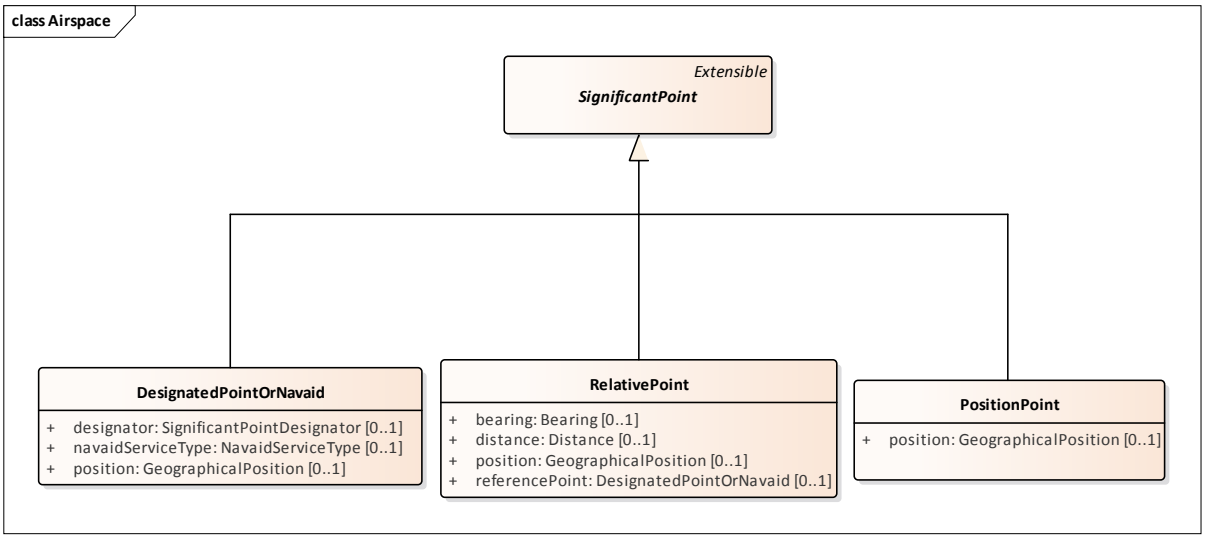
Classes that are marked “abstract” are often used as the base for inheritance, but can never be physically instantiated in XML form: only their concrete descendants can be instantiated.

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 modelling.
- 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 37 - Abstract classes are allowed
Best Practice 38 - Root classes are forbidden
Best Practice 39 - Leaf classes must be justified



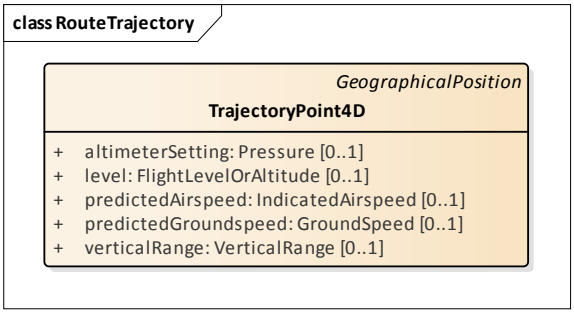


Figure 4 - Inheritance Notations

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 “GeographicalPosition” of the “position” attribute in Figure 4. For relations, the datatype is the type of the class on the target end of the relation. Primitive datatypes are available for use only in the “Base” packages: within the “Flight” packages it is an error to declare an attribute of primitive type. The full set of allowed primitive types is:

- int
- string
- boolean
- double
- date
- time
- dateTime
-
- duration

Best Practice 40 - Set of primitive types is restricted
Best Practice 41 - Primitive types used only in “Base” packages

3.5 Constraints on Boolean Values

Because boolean elements have only two values, “true” or “false”, their semantics depend on the context of their use and might be 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.

Best Practice 42 – Use single valued enumerations to represent irreversible states
Best Practice 43 – 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 Flight 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 5 illustrates a choice class ColourChoice, and shows that attributes (otherColour) can be intermingled with relations (colourCode) as alternatives.
- «enumeration»
  Represents a selection from a set of named string values.
- <<deprecated>>

  Indicates that the Element, attribute or property is deprecated and is slated for removal or at a later release. Please see Sectio 3.17 for details.

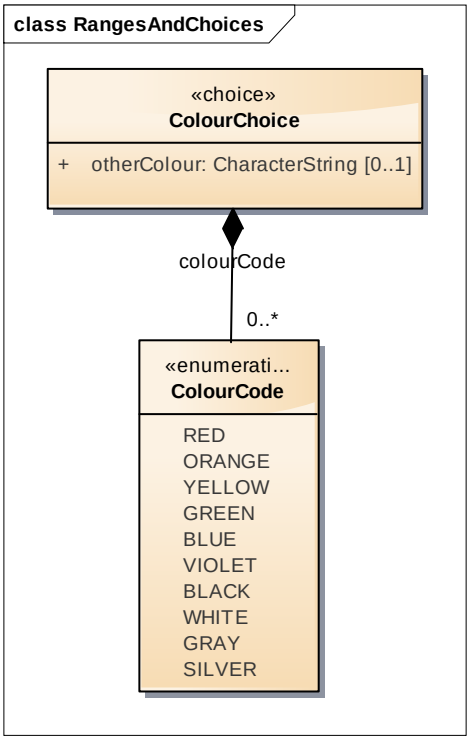


Figure 5 - Use of «choice» Stereotype

Best Practice 44 - Set of stereotypes is restricted
Best Practice 45 - "Normal" Flight classes have no stereotype
Best Practice 46 - Use «choice» to show alternative properties
Best Practice 47 - Multiplicity 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.

Best Practice 48 - Enumeration values such as “OTHER” or “UNKNOWN” are discouraged
Best Practice 49 - Use an "otherText" alternative if necessary

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 modellers, 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 Operational Data Description, but since some container classes do not have direct relatives in the directory, the data modellers need to supply definitions for these elements.

The documentation for the model elements should also include the origin of the reference, if known. For example, Figure 6 contains the reference [FIXM], which means that the definition was specifically developed for the FIXM Model. Other examples found throughout the FIXM model include[AIXM 5.1], which indicate that the element was adapted from AIXM model version 5.1, and ICAO Doc 4444, indicating that the concept and definition was derived from ICAO 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

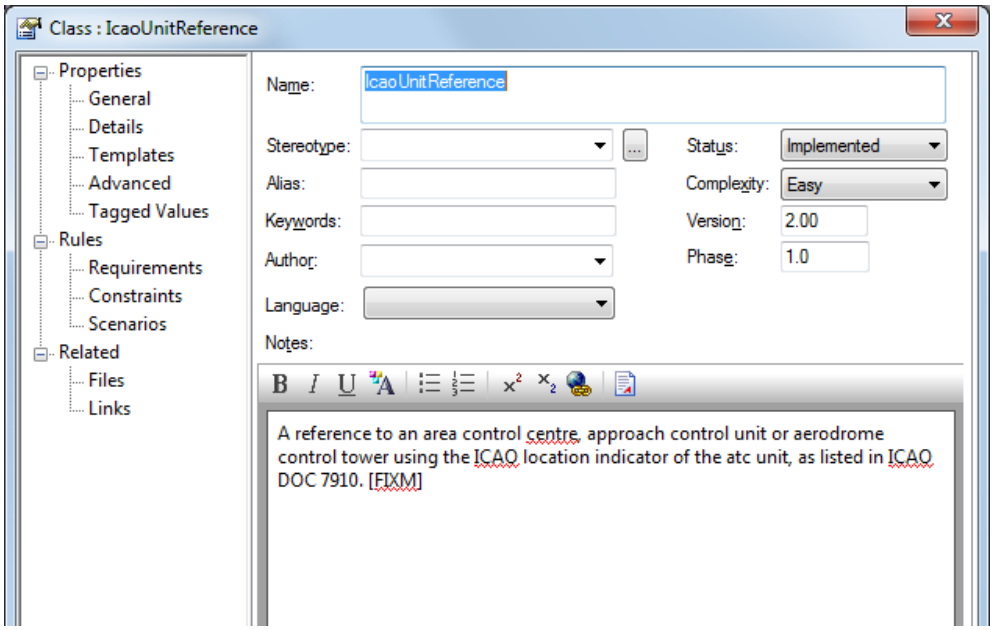


Figure 6 - Example of element documentation

Best Practice 50 - All model components should be documented

Best Practice 51 – Operation Data Description comments are considered adequate documentation

Best Practice 52 - Add extra documentation as needed for clarity

Best Practice 53 - Apply documentation where data is defined

Best Practice 54 - Documentation should not refer to other model objects

Best Practice 55 - Package documentation should describe the theme of the package

3.9 Constraints

For many data items, the FIXM Operation Data Description 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 56 - Use Constraints to capture limitations on data values

Best Practice 57 - 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 7 illustrates how to set these in the target end of relations.

If the “ordered” checkbox is selected, then the list is presumed to be sorted according to the natural ordering of its contained types. In addition to the ordered property, which provides visual information in the Logical Model, it is necessary to include a clue for the automated schema generator to generate a sequence number attribute in the class that is being ordered. This is described in Section 3.17.

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 58 - Indicate ordering of attributes and relationships

Best Practice 59 - Indicate when attributes and relationships contain duplicate values

Best Practice 60 - When ordering of structured types is specified, the documentation must make the ordering relation explicit

Best Practice 61 - By default an order relation is ascending

Best Practice 62 - If an order is descending, this must be stated explicitly in documentation

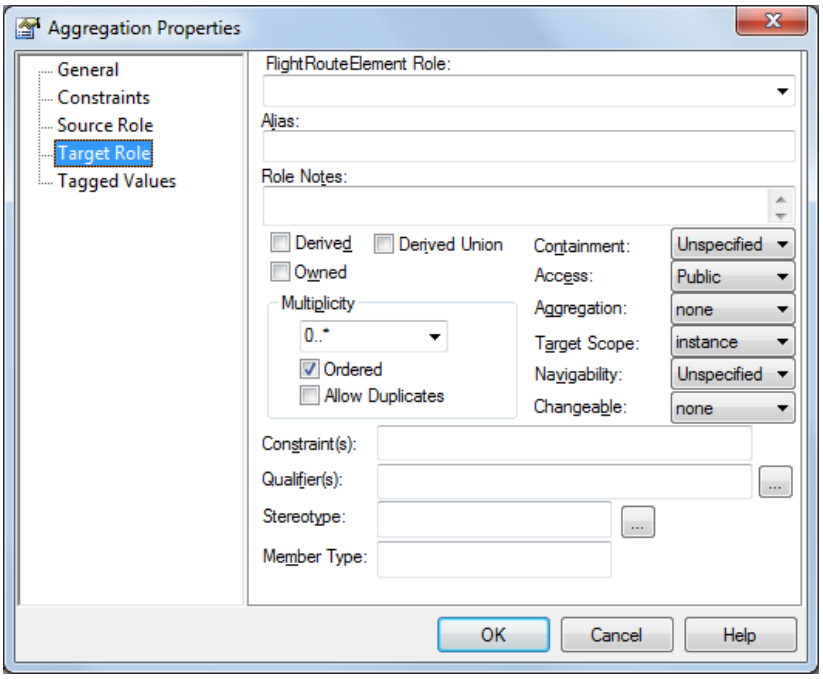


Figure 7 - Specifying Ordering of a Relationship

3.11 Requirements

The “Requirements” section of the Enterprise Architect data may contain one or more references back to the Operational Data Description, as shown in Figure 8. In Enterprise Architect, neither attributes nor relations have storage for requirements, so when Operational Data Description 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 operational data description entry.

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

Best Practice 63 - Use Requirements of type TRACE to map operational data description entries
Best Practice 64 - Relations and attributes map to their containing classes

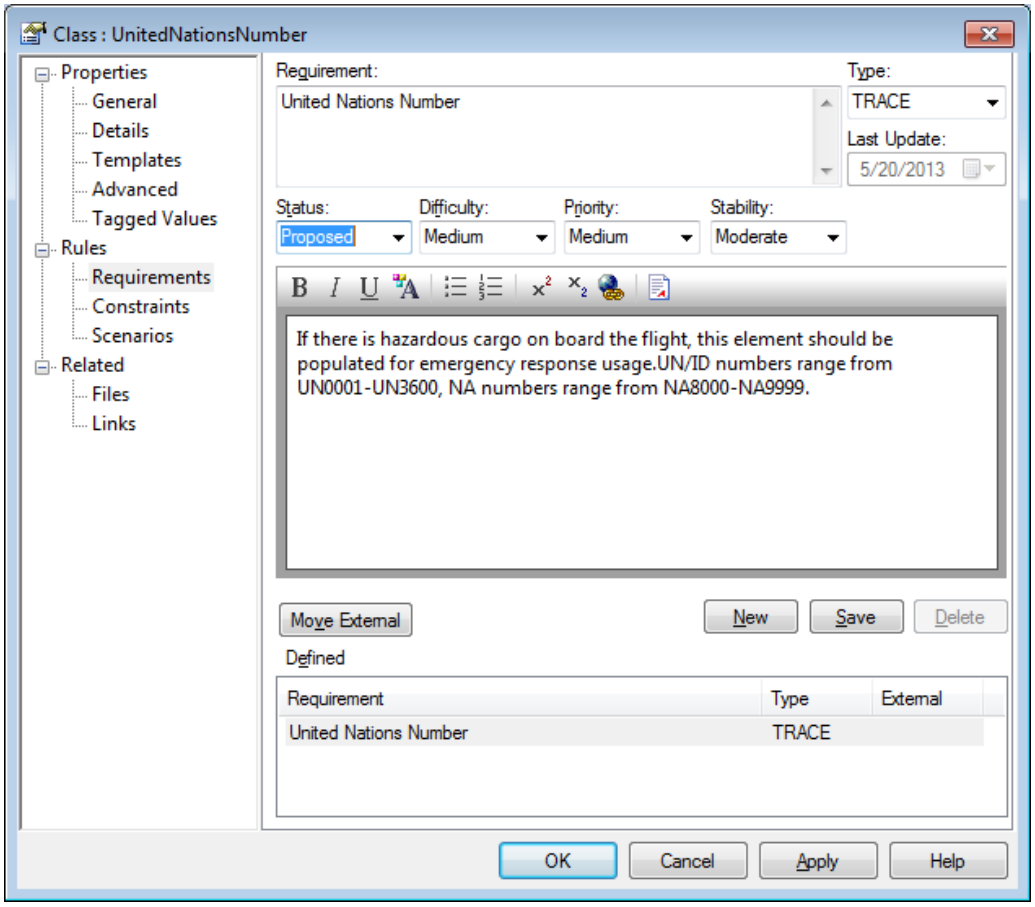


Figure 8 - Requirements and Files

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 9. 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 65 - Version of attributes and relations map to containing type

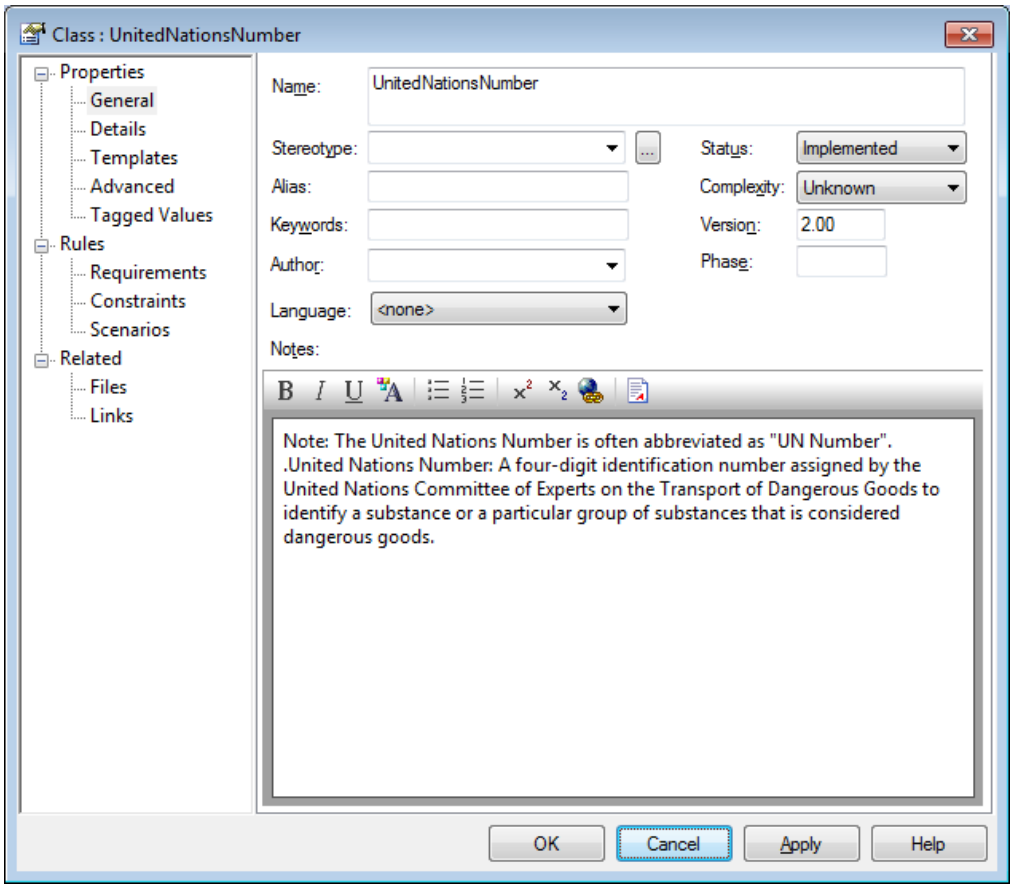


Figure 9 - Element Status, Version, Authorship

3.13 Authorship

Elements of FIXM are presumed to be “authored” by the entire data modelling team and not by an individual, so the “Author” field should be blank as shown in Figure 9. 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 66 - Model elements should not be related to individual authors

3.14 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 exceptions to this rule. In Figure 10 the type AerodromeName contradicts this best practice because it adds neither information nor pattern to the TextName type it extends, however since it represents an important concept; 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. In addition, it also provides the flexibility to modify the type definition independent of the type it extends. 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 67 - Avoid empty extensions where they add no information

Best Practice 68 - Use empty extensions if they clarify intent

Best Practice 69 - Model elements should not be related to individual authors

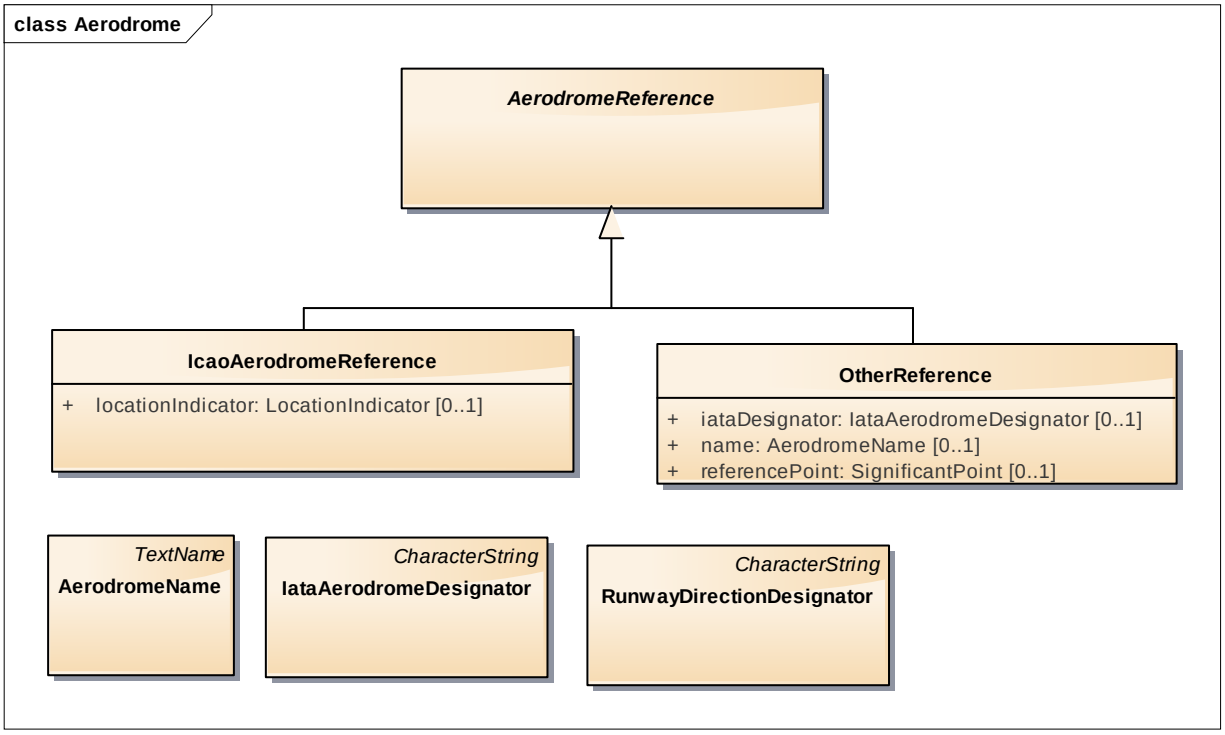


Figure 10 - Empty Extension Emphasizes Important Concept

3.15 Constraining Property Values

Most of the data entries of the FIXM Operational Data Description contain some description of constraints on the “Basic” data type, and it is part of the data modelling 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 values 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 11 illustrates the steps in constraining attribute “ShippingInformation.supplementaryInformation” 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 70 - Use Constraints to capture data restrictions from the operational data description

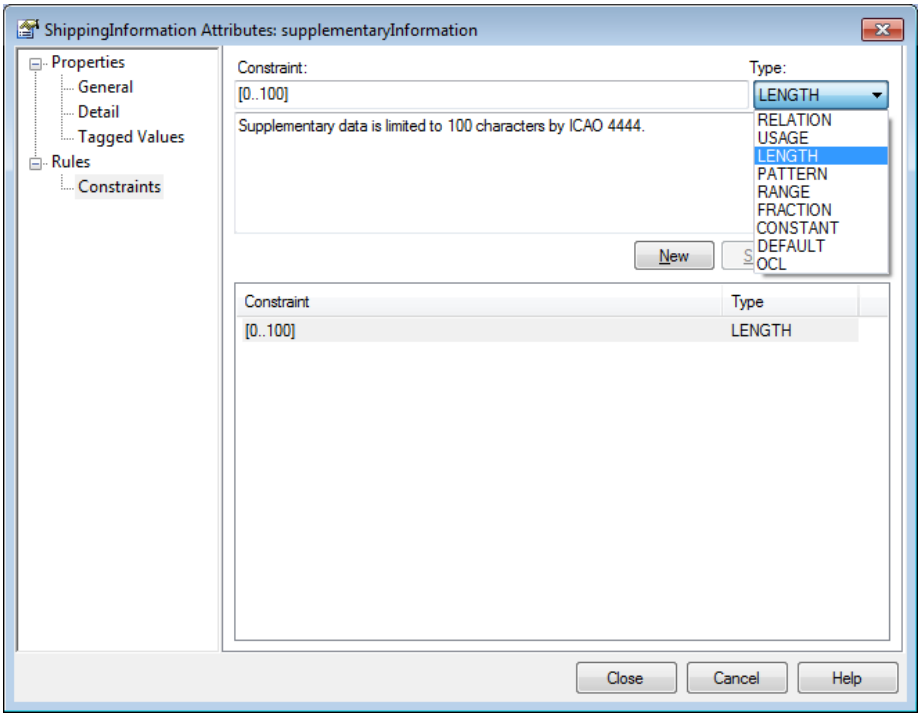


Figure 11 - Constraining an Attribute's Length

3.16 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 12.

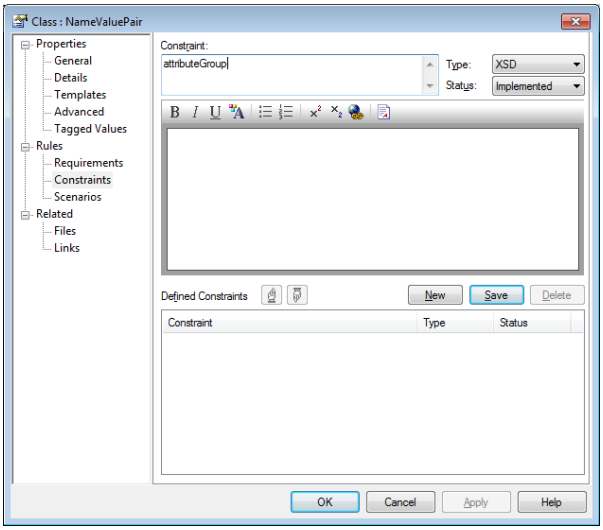


Figure 12 - Specifying an XSD Constraint

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

Table 2 – XSD Constraints guides

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'.
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.
Ordered	Properties	Guides a generation of an optional seqNum attribute in the XSD to allow specifying item order in the sequence.
Xlinks	Properties	Guides a generation of an optional attribute group <attributeGroup ref="*xlink:simpleAttrs*" /> in order to allow referencing externally defined data such as from AIXM.

3.17 Deprecation

Deprecation of elements and properties can be accomplished by placing a <<deprecated>> stereotype by each item. The stereotype labelling of an element and property is demonstrated in Figure 13.

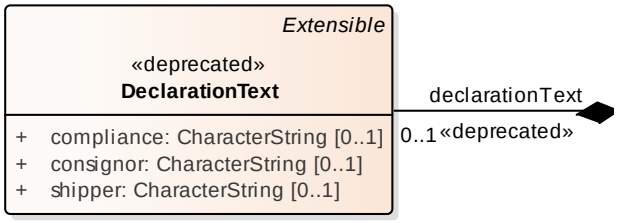


Figure 13 - Specifying Deprecated stereotype

In addition, each item must include tagged values that specify deprecation details. These are described in Table 4:

Requirement	Description
Rationale	The reason for the deprecation, such as a replacement or a particular capability being no longer supported.
Replacement	A reference to other model element(s) that should be used instead, if applicable.
Deprecation version	The version in which the deprecation occurs.
Deletion version	The version in which the deletion is planned to occur.

Table 3 – Deprecation Tags

For each model element deprecated in FIXM:

- Field “Rationale” will include:
  - A reference to the FIXM CR having requested the deprecation
  - As appropriate, relevant text from the FIXM CR - e.g. text extracted from the CR’s “Motivation” section – so that the justification for the deprecation is self-contained.
- Field “Replacement”, if applicable, will include:
  - The path to the corresponding FIXM model element replacing the deprecated element. Example: Fixm.Flight.[…], similar to the tracing captured in the ODD.

In Enterprise Architect, each item must contain Tagged values containing deprecation details as described in Figure 14

Figure 14 - Specifying Deprecated Tagged Values

For Each XML property and type deprecated in FIXM, the generated XSD schema will include deprecation information under the XML annotation tag, as shown in Figure 15.

```
<annotation>
    <documentation>
      <deprecated>
        <rationale>FIXM CR## - this element is deprecated because… </rationale>
        <replacement>as appropriate – put ‘N/A’ if no replacement is available </replacement>
        <deprecationVersion>4.1.0</deprecationVersion>
        <deletionVersion>5.0.0</deletionVersion>
      </deprecated>
    </documentation>
</annotation>
```

Figure 15 - Deprecated annotation in XSD

For entire elements/classes that are being deprecated, it is necessary to provide deprecation information for every element, property and association (connector) that is being deprecated.

4 Modelling 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 or stakeholder-specific information to accommodate ATM procedures of individual countries and regions. Please refer to the "FIXM Development Guidelines" document for further principles on FIXM development.

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. In some situations this governing body may also authorize development of Verified extension. Such extension may contain elements candidates for the FIXM core package or elements that are needed for temporary international usage (for example during transition to a fully FF-ICE compatible environment). By contrast, regional or system specific extension models may be developed by any organization, as needed to provide additional flight data.

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 when creating 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 flight packages, or to classes defined in the FIXM base package. 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 71 - Extensions may refer to their own data elements.

Best Practice 72 – Extensions may refer to data elements in the FIXM core packages.

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 16 illustrates the use of the prefix "Nas" to distinguish the FAA National Airspace System (NAS) extension arrival class from the core Arrival class, because the similarity of names is likely to lead to confusion.

Best Practice 73 - Use a short prefix to distinguish extension from core classes.

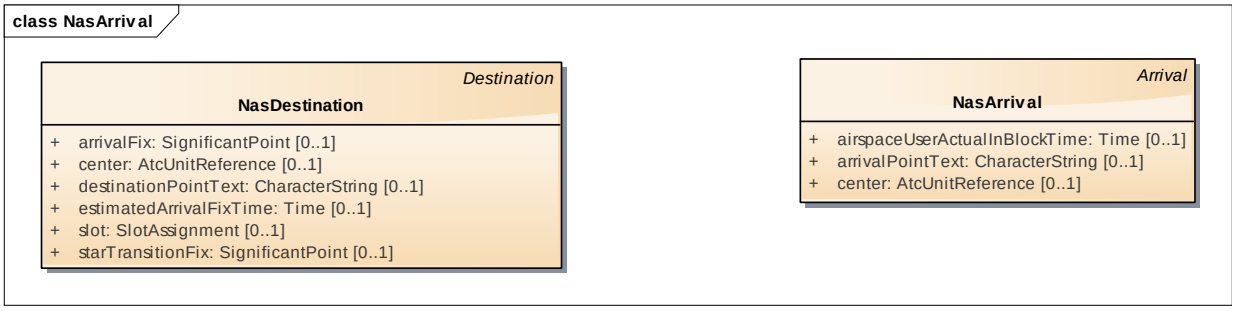


Figure 16 - Use of prefixes to qualify names

4.2.4 Extension Strategy 1: Use of UML Inheritance

Each class 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 17 illustrates this pattern: the US extension requires the definition of an Arrival class that contains additional attributes, such as the arrival center. In this figure, we see that the “NasArrival” class is defined as an extension of the “Arrival” class from the core packages, adding its own “center” property. In constructing a US extension message, the “NasArrival” class can be used anywhere that the “Arrival” class is valid. Applications that expect the US extensions will use either the “Arrival” or NasArrival fields, but applications that are not expecting a US extension will use only the Arrival fields.

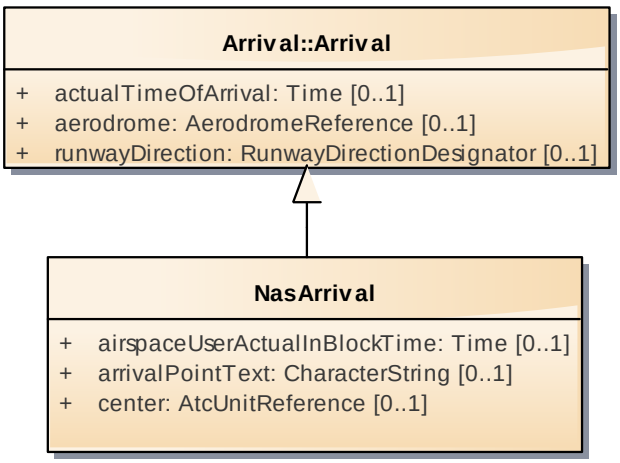


Figure 17 – Extending a base class to change constraints

Best Practice 74 – Add data to core classes using UML inheritance.

Of course, this pattern can be applied to other, higher level classes.

Figure 18 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 75 - Extend FIXM Core to define new flight data

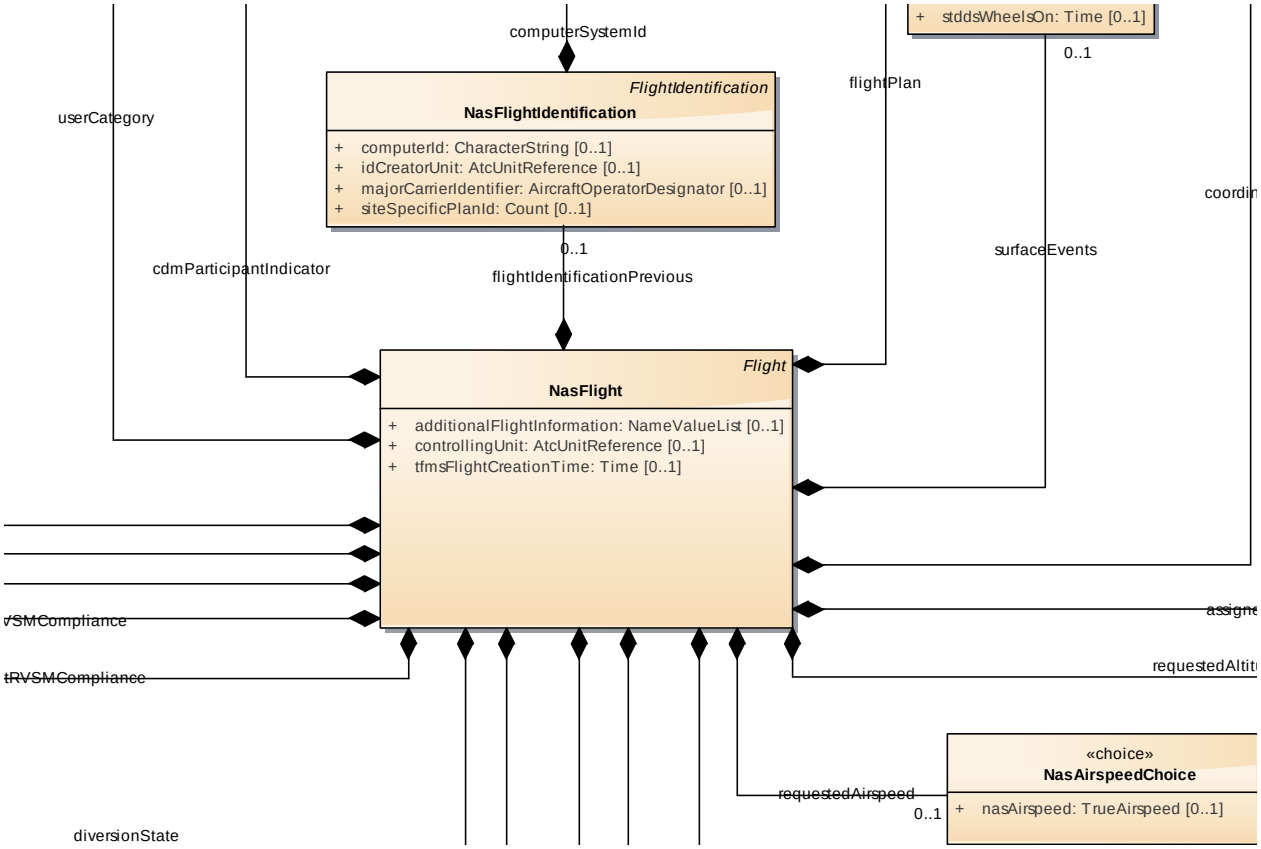


Figure 18 - Extending a Core Flight

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 US extension uses NAS route text (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.

Best Practice 76 - 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 19.

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.

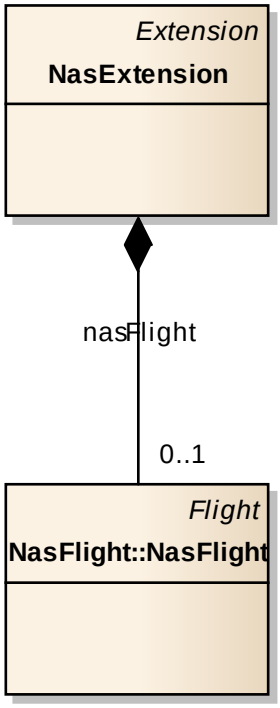


Figure 19 - Defining an Extension Subclass

4.2.7 Extension Strategy 3. Use of the Extensible and Extension classes

This approach introduces a new abstract class *Extensible* that contains sequence of zero or more attributes named *extension* whose content must derive from class *Extension*. The multiplicity of attribute *extension* of class *Extensible* is 0..*. Any class that has a potential to be extended must derive from the class *Extensible*, and any class extending it must derive from the class *Extension*.

Different extensions that need to extend the same core class would employ different *extension* elements of the class, thus neither, either, or both of the extension elements could be included in an XML document with no dependencies between them, and all are compatible with core. The goal of this approach is allowing data consumers to easily ignore Extension data. Documentation will be produced for guidance (including explicit examples) on the use of XSLTs to remove unwanted Extension data in the FIXM implementation guidance document.

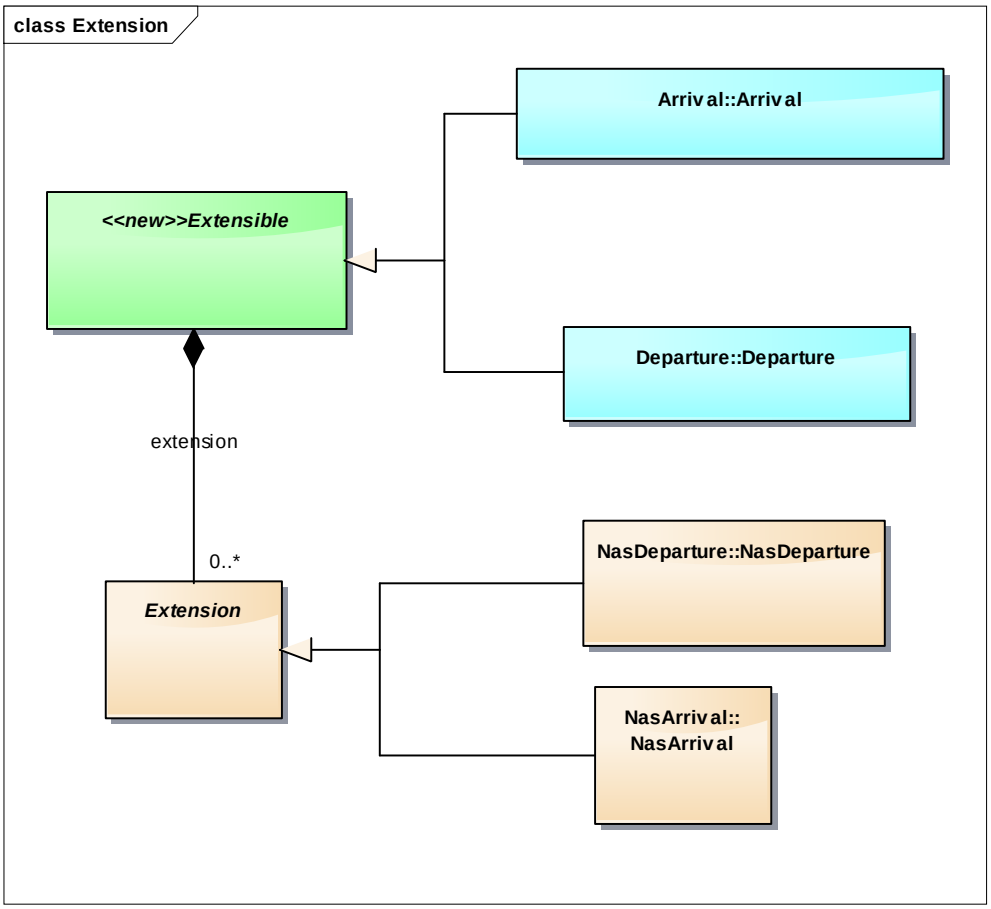


Figure 20 – Uses of Extensible and Extension classes

A question arises: what classes in the core should generalise *Extensible*. The following rules are approved:

A class in core FIXM should generalise class *Extensible* **unless**
- The class translates to a Complex Type in the XSD Schema
- The class is a choice
- The class is an enumeration
- The class is already a generalisation of another class.

4.2.8 Comparison of Strategies

It is important to compare the three 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. However that strategy results in extensions that are incompatible with the FIXM. If the receiver only has a Core FIXM schema and receives XML created using an Extension schema, the entire XML cannot be validated.

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. The duplication arises when multiple paths exist to similar data. Because the classes are grouped in a separate area and include additional associations aside from the core model associations, the data may be represented both in the core location, and in the extended location accessible from multiple paths.

Strategy 3 was created to avoid the issues associated with either of the aforementioned strategies. The goal was twofold: Allowing multiple extensions of the same class to be used together and allowing the user to ignore extension data easily, while still being able to process the core portion of the data. It is based on this information, it is recommended to use Strategy 3 for defining FIXM extensions.

4.3 Extension Projects

Developing a FIXM extension requires that it be modelled using Enterprise Architect, the standard UML modelling tool for FIXM. Figure 21 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_<version>.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.

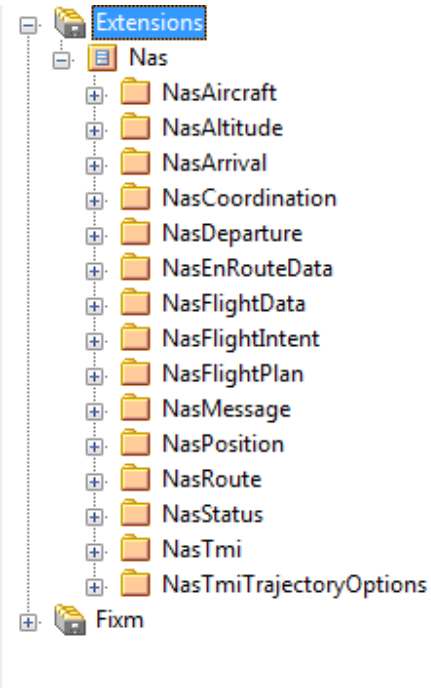


Figure 21 - Structure of an Enterprise Architect Extension Project

After the extension model has been created, it can be processed through the FIXM Modelling Workbench in order to generate the XSD. The FIXM Workbench is accessible at https://fixm.aero/tools.pl.

Appendix A FIXM Copyright Notice

Best Practices Summary