SDMX

INFORMATION MODEL:

UML CONCEPTUAL DESIGN

(VERSION 2.0)

November 2005

Contents

Change History

Version 1.0 – initial release September 2004.

Version 2.0 – release November 2005

Major functional enhancements by addition of new packages:

- Metadata Structure Definition
- Metadata Set
- Hierarchical Code Scheme
- Data and Metadata Provisioning
- Structure Set and Mappings
- Transformations and Expressions
- Process and Transitions

Re-engineering of some SDMX Base structures to give more functionality:

- Item Scheme and Item can have properties – this gives support for complex hierarchical code schemes (where the property can be used to sequence codes in scheme), and Item Scheme mapping tables (where the property can give additional information about the map between the two schemes and the between two Items)
- revised Organisation pattern to support maintained schemes of organisations, such as a data provider
- modified Component Structure pattern to support identification of roles played by components and the attachment of attributes
- change to inheritance to enable more artefacts to be identifiable and versionable

Introduction of new types of Item Scheme:

- Object Type Scheme to specify object types in support of the Metadata Structure Definition (principally the object types (classes) in this Information Model)
- Type Scheme to specify types other than object type
- A generic Item Scheme Association to specify the association between Items in two or more Item Schemes, where such associations cannot be described in the Structure Set and Transformation.

The Data Structure Definition is introduced as a synonym for Key Family, though the term Key Family is retained and used in this specification.

Modification to Key Family (Data Structure Definition) to

- align the cross sectional structures with the functionality of the schema
- support key family extension (i.e. to derive and extend a key family from another key family), thus supporting the definition of a related “set” of key families
- distinguish between data attributes (which are described in a key family) from metadata attributes (which are described in a metadata structure definition)
- attach data attributes to specific identifiable artefacts (formally this was supported by attachable artefact)

Domain Category Scheme re-named Category Scheme to better reflect the multiple usage of this type of scheme (e.g. subject matter domain, reporting taxonomy).

Concept Scheme enhanced to allow specification of the representation of the Concept. This specification is the default (or core) representation and can be overridden by a construct that uses it (such as a Dimension in a Key Family).

Revision of cross sectional data set to reflect the functionality of the version 1.0 schema.

Revision of Actors and Use Cases to reflect better the functionality supported.

1 INTRODUCTION

This document is not normative, but provides a detailed view of the information model on which the normative SDMX specifications are based. Those new to the UML notation or to the concept of key families may wish to read the appendixes in this document as an introductory exercise.

1.1 Related Documents

This document is one of three documents concerned with the SDMX Information Model. The complete set of documents is:

SDMX INFORMATION MODEL: UML CONCEPTUAL DESIGN (this document)

This document comprises the complete definition of the information model, with the exception of the registry interfaces. It is intended for technicians wishing to understand the complete scope of the SDMX technical standards in a syntax neutral form.

SDMX REGISTRY SPECIFICATION: LOGICAL INTERFACES

This document provides the logical specification for the registry interfaces, including subscription/notification, registration/submission of data and metadata, and querying.

SDMX IMPLEMENTORS GUIDE

This document explains the structures in the model in high level diagrammatic form and maps these diagrams to the class diagrams in the model. In addition it gives worked examples of the structures. It is intended for technicians wishing to gain an overall understanding of the structures of the model in a more informal and less complete form than the UML Conceptual Design.

1.2 Modelling Technique and Diagrammatic Notes

The modelling technique used for the SDMX Information Model (SDMX-IM) is the Unified Modelling Language (UML). An overview of the constructs of UML that are used in the SDMX-IM can be found in the Appendix "A Short Guide to UML in the SDMX Information Model"

UML diagramming allows a class to be shown with or without the compartments for one or both of attributes and operations (sometimes called methods). In this document the operations compartment is not shown as there are no operations.

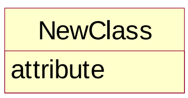


Figure 1 Class with operations suppressed

In some diagrams for some classes the attribute compartment is suppressed even though there may be some attributes. This is deliberate and is done to aid clarity of the diagram. The rules used are:

- The attributes will always be present on the class diagram where the class is defined and its attributes and associations are defined.

- On other diagrams, such as inheritance diagrams, the attributes may be suppressed from the class for clarity.

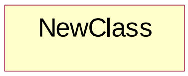


Figure 2 Class with attributes also suppressed

Note that, in any case, attributes inherited from a super class are not shown in the sub class.

The following table structure is used to in the definition of the classes, attributes, and associations.

Class	**Feature**	**Description**
`ClassName`		
	`attributeName`	.
	`associationName`	
	`+roleName`	

The content in the “Feature” column comprises or explains one of the following structural features of the class:

- Whether it is an abstract class. Abstract classes are shown in *`italic Courier`* font

- The superclass this class inherits from, if any

- The sub classes of this class, if any

- Attribute – the `attributeName` is shown in `Courier` font

- Association – the `associationName` is shown in `Courier` font. If the association is derived from the association between super classes then the format is `/associationName`

- Role – the `+roleName` is shown in `Courier` font

The Description column provides a short definition or explanation of the Class or Feature. UML class names may be used in the description and if so, they are presented in normal font with spaces between words. For example the class `CodeList` will be written as Code List.

1.3 Overall Functionality

1.3.1 Information Model Packages

The SDMX Information Model (SDMX-IM) is a conceptual metamodel from which syntax specific implementations are developed. The model is constructed as a set of functional packages which assist in the understanding, re-use and maintenance of the model.

In addition to this, in order to aid understanding each package can be considered to be in one of three conceptual layers:

- the SDMX Base layer comprises fundamental building blocks which are used by the Structural Definitions layer and the Reporting and Dissemination layer
- the Structural Definitions layer comprises the definition of the structural artefacts needed to support data and metadata reporting and dissemination
- the Reporting and Dissemination layer comprises the definition of the data and metadata containers used for reporting and dissemination

In reality the layers have no implicit or explicit structural function as any package can make use of any construct in another package.

1.3.2 Version 1.0

In version 1.0 the metamodel supported the requirements for:

- Key family definition including (domain) category scheme, (metadata) concept scheme, and code list

- Data and related metadata reporting and dissemination

The SDMX-IM comprises a number of packages. These packages act as convenient compartments for the various sub models in the SDMX-IM. The diagram below shows the sub models of the SDMX-IM that were included in the version 1.0 specification.

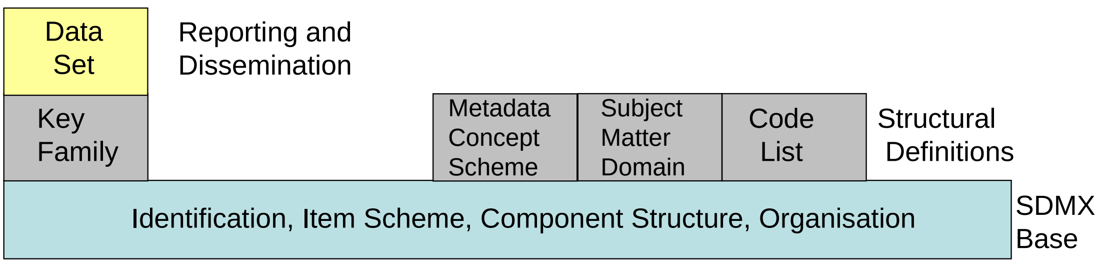


Figure 3: SDMX Information Model Version 1.0 package structure

1.3.3 Version 2.0

The version 2.0 model extends the functionality of version 1.0. principally in the area of metadata, but also in various ways to define structures to support data analysis by systems with knowledge of cube type structures such as OLAP[1] systems. The following packages have been added at version 2.0

- Metadata structure definition
- Metadata set
- Hierarchical code scheme
- Cube definition
- Data and metadata provisioning
- Transformations and expressions

Furthermore, the synonym Data Structure Definition is assigned to the Key Family as these two terms are used in various communities and they are synonymous. The term Key Family is used in this document.

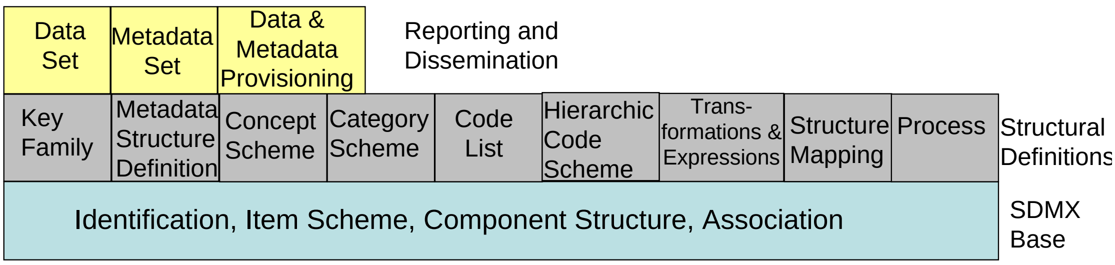


Figure 4 SDMX Information Model Version 2.0 package structure

Additional packages that are specific to a registry based scenario can be found in the Specification of Registry Interfaces. For information these are shown on the diagram below and comprise:

- Subscription and Notification
- Registration
- Discovery

Note that the data and metadata required for registry functions are not confined to these three packages, and the registry also makes use of the other packages in the Information Model.

---

[1] OLAP: On line analytical processing

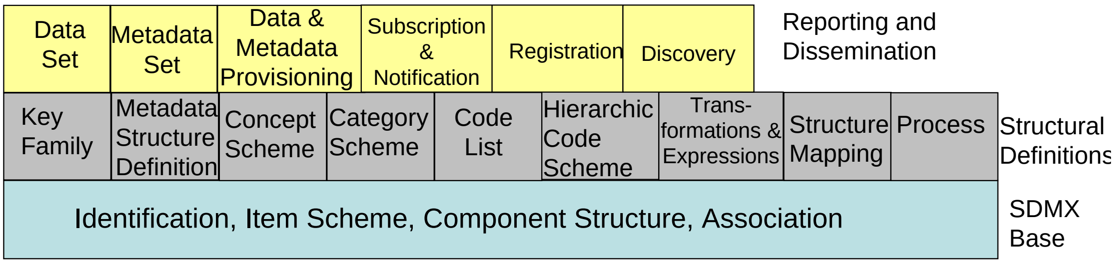


Figure 5: SDMX Information Model Version 2.0 package structure including the registry

2 ACTORS AND USE CASES

2.1 Actors and Use Cases

In order to develop the data models it is necessary to understand the functions to be supported resulting from the requirements definition. These are defined in a use case model. The use case model comprises actors and use cases and these are defined below.

Actor

"*An actor defines a coherent set of roles that users of the system can play when interacting with it. An actor instance can be played by either an individual or an external system*"

Use case

"A use case defines a set of use-case instances, where each instance is a sequence of actions a system performs that yields an observable result of value to a particular actor"

The overall intent of the model is to support data and metadata reporting, dissemination, and exchange in the field of aggregated statistical data and related metadata. In order to achieve this, the model needs to support three fundamental aspects of this process:

- Maintenance of structural and provisioning definitions
- Data and metadata publishing (reporting), and consuming (using)
- Access to data, metadata, and structural and provisioning definitions

This document covers the first two aspects, whilst the document on the Registry logical model covers the last aspect.

2.2 Use Case Diagrams

2.2.1 Maintenance of Structural and Provisioning Definitions

2.2.1.1 Use cases

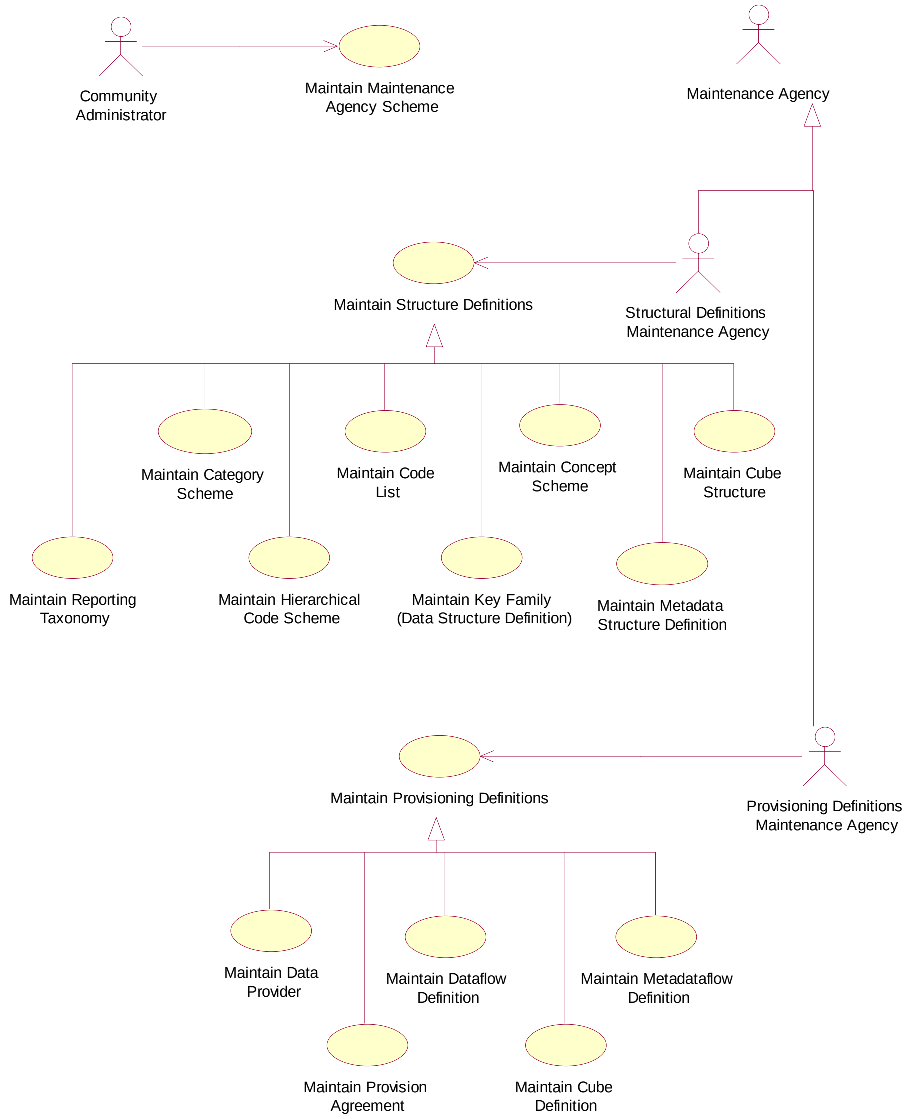


Figure 6 Use cases for maintaining data and metadata structural and provisioning definitions

2.2.1.2 Explanation of the Diagram

In order for applications to publish and consume data and metadata it is necessary for the structure and permitted content of the data and metadata to be defined and made available to the applications, as well as definitions that support the actual process of publishing and consuming. This is the responsibility of a Maintenance Agency.

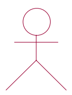
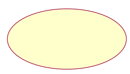
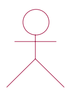
All maintained artefacts are maintained by a Maintenance Agency. For convenience the Maintenance Agency actor is sub divided into two actor roles:

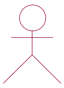
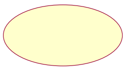
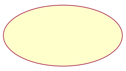
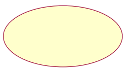
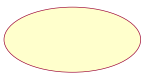
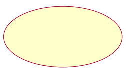
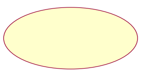
- maintaining structural definitions
- maintaining provisioning definitions

Whilst both these functions may be carried out by the same person, or at least by the same maintaining organization, the purpose of the definitions is different and so the roles have been differentiated: structural definitions define the format and permitted content of data and metadata when reported or disseminated, whilst provisioning definitions support the process of reporting and dissemination (who reports what to whom, and when).

In a community based scenario where at least the structural definitions may be shared, it is important that the scheme of maintenance agencies is maintained by a responsible organization (called here the Community Administrator).

2.2.1.3 Definitions

Actor	Use Case	Description
		Responsible organisation that administers structural definitions common to the community as a whole.
		Creation and maintenance of the scheme of maintenance agencies.
		Responsible agency for maintaining structural artefacts such as code lists, concept schemes, key family structural definitions, metadata structure definitions, and data and metadata provisioning artefacts such as data

Actor	Use Case	Description
		providers and dataflow definitions. sub roles are: Structural Definitions Maintenance Agency Provisioning Definitions Maintenance Agency
 Structural Definitions Maintenance Agency		Responsible for maintaining structural definitions.
	 Maintain Structure Definitions	The maintenance of structural definitions. This use case has sub class use cases for each of the structural artefacts that are maintained.
	 Maintain Code List  Maintain Concept Scheme  Maintain Category Scheme  Maintain Key Family (Data Structure Definition)  Maintain Metadata Structure Definition	Creation and maintenance of the key family (data structure definition), metadata structure definition, and cube structure, and the supporting artefacts that they use, such as code list and concept scheme.

Actor	Use Case	Description
	Maintain Cube Structure Maintain Hierarchical Code Scheme Maintain Reporting Taxonomy	
Provisioning Definitions Maintenance Agency		Responsible for maintaining data and metadata provisioning definitions.
	Maintain Provisioning Definitions	The maintenance of provisioning definitions. This use case has sub class use cases for each of the structural artefacts that are maintained.
	Maintain Data Provider Maintain Dataflow Definition Maintain Metadataflow Definition	Creation and maintenance of the artefacts that support the definition of data and metadata provisioning, such as the list of data providers, dataflow definitions, cube definitions, and the provision agreements that link the data providers with the dataflow and metadata flow definitions.

Actor	Use Case	Description
	Maintain Cube Definition Maintain Provision Agreement	

Figure 7: Table of Actors and use Cases for Maintenance of Structural and Provisioning Definitions

2.2.2 Publishing and Using Data and Metadata

2.2.2.1 Use Cases

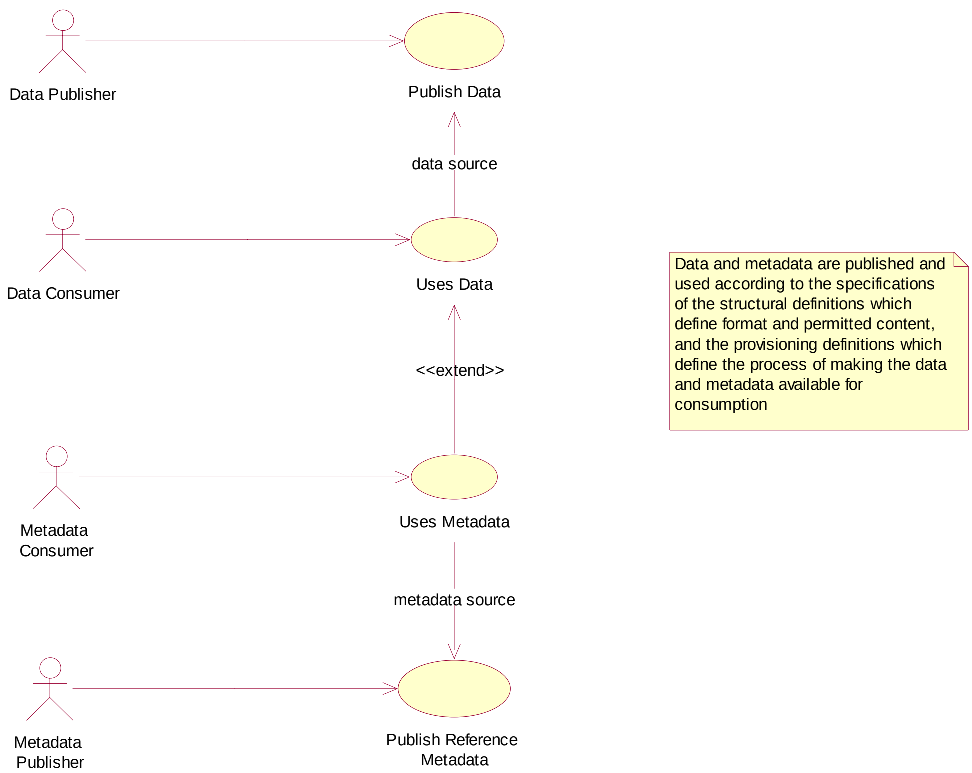


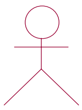
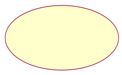
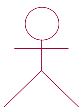
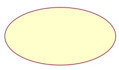
Figure 8: Actors and use cases for data and metadata publishing and consuming

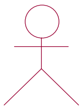
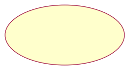
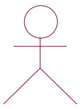
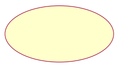
2.2.2.2 Explanation of the Diagram

Note that in this diagram "publishing" data and metadata is deemed to be the same as "reporting" data and metadata. In some cases the act of making the data available fulfils both functions. Aggregated data is published and in order for the Data Publisher to do this and in order for consuming applications to process the data and metadata its structure must be known. Furthermore, consuming applications may also require access to (reference) metadata in order to present this to the Data Consumer so that the data is better understood. As with the data, the reference metadata also needs to be formatted in accordance with a maintained structure. The Data Consumer and Metadata Consumer cannot use the data or metadata unless it is "published" and so there is a "data source" or "metadata source" dependency between the "uses" and "publish" use cases.

In any data and metadata publishing and consuming scenario both the publishing and the consuming applications will need access to maintained Provisioning Definitions. These definitions may be as simple as who provides what data and metadata to whom, and when, or it can be more complex with constraints on the data and metadata that can be provided by a particular publisher, and, in a data sharing scenario where data and metadata are "pulled" from data sources, details of the source.

2.2.2.3 Definitions

Actor	Use Case	Description
 Data Publisher		Responsible for publishing data according to a specified key family (data structure) definition, and relevant provisioning definitions.
	 Publish Data	Publish a data set. This could mean a physical data set or it could mean to make the data available for access at a data source such as a database that can process a query.
 Data Consumer		The user of the data. It may be a human consumer accessing via a use interface, or it could be an application such as a statistical production system.
	 Uses Data	Use data that is formatted according to the structural definitions and made available according to the provisioning definitions. Data are often linked to metadata that may reside in a different location and be published and maintained independently.

Actor	**Use Case**	**Description**
 Metadata Publisher		Responsible for publishing reference metadata according to a specified metadata structure definition, and relevant provisioning definitions.
	 Publish Reference Metadata	Publish a reference metadata set. This could mean a physical metadata set or it could mean to make the metadata available for access at a metadata source such as a metadata repository that can process a query.
 Metadata Consumer		The user of the metadata. It may be a human consumer accessing via a use interface, or it could be an application such as a statistical production or dissemination system.
	 Uses Metadata	Use metadata that is formatted according to the structural definitions and made available according to the provisioning definitions.

3 SDMX BASE PACKAGE

3.1 Introduction

The constructs in the SDMX Base package comprise the fundamental building blocks that support many of the other structures in the model. For this reason, many of the classes in this package are abstract (i.e. only derived sub-classes can exist in an implementation).

The motivation for establishing the SDMX Base package is as follows:

- It is accepted "Best Practise" to identify fundamental archetypes occurring in a model
- identification of commonly found structures or "patterns" leads to easier understanding
- identification of patterns encourages re-use

Each of the class diagrams in this section views classes from the SDMX Base package from a different perspective. There are detailed views of specific patterns, plus overviews showing inheritance between classes, and relationships amongst classes.

3.2 Identification, Versioning, and Maintenance

3.2.1 Class Diagram

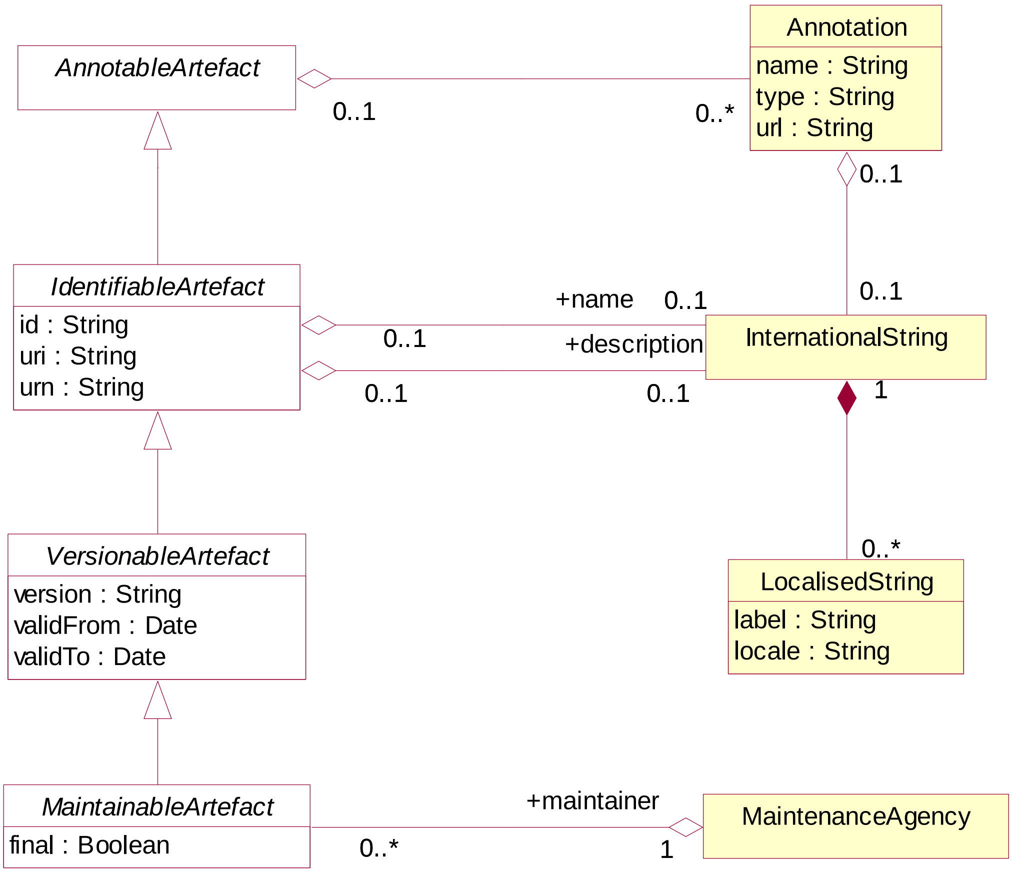


Figure 9 SDMX Identification, maintenance and versioning

3.2.2 Explanation of the Diagram

3.2.2.1 Narrative

This group of classes forms the nucleus of the administration facets of SDMX objects. They provide features which are reusable by derived classes to support horizontal functionality such as identity, versioning etc.

All classes derived from the abstract class *`AnnotableArtefact`* may have Annotations (or notes): this supports the need to add notes to all SDMX-ML elements. The Annotation is used to convey extra information to describe any SDMX construct. This information may be in the form of a URL reference and / or a multilingual text (represented by the association to `InternationalString`).

The *IdentifiableArtefact* is an abstract class that comprises the basic attributes needed fir identification. Concrete classes based on *IdentifiableArtefact* all inherit the ability to be uniquely identified. They also inherit the ability to carry annotations. In addition, the +description and +name roles support multilingual descriptions and names for all objects based on *IdentifiableArtefact*. The InternationalString supports the representation of a description in multiple locales (locale is similar to language but includes geographic variations such as Canadian French, US English etc.). The *LocalisedString* supports the representation of a description in one locale.

VersionableArtefact is an abstract class which inherits from *IdentifiableArtefact* and adds versioning ability to all classes derived from it.

MaintainableArtefact further adds the ability for derived classes to be maintained via its association to *MaintenanceAgency*. It is possible to define whether the artefact is draft or final with the final attribute.

The inheritance chain from *AnnotableArtefact* through to *MaintainableArtefact* allows SDMX classes to inherit the features they need, from simple annotation, through identity, to versioning and maintenance.

3.2.2.2 Definitions

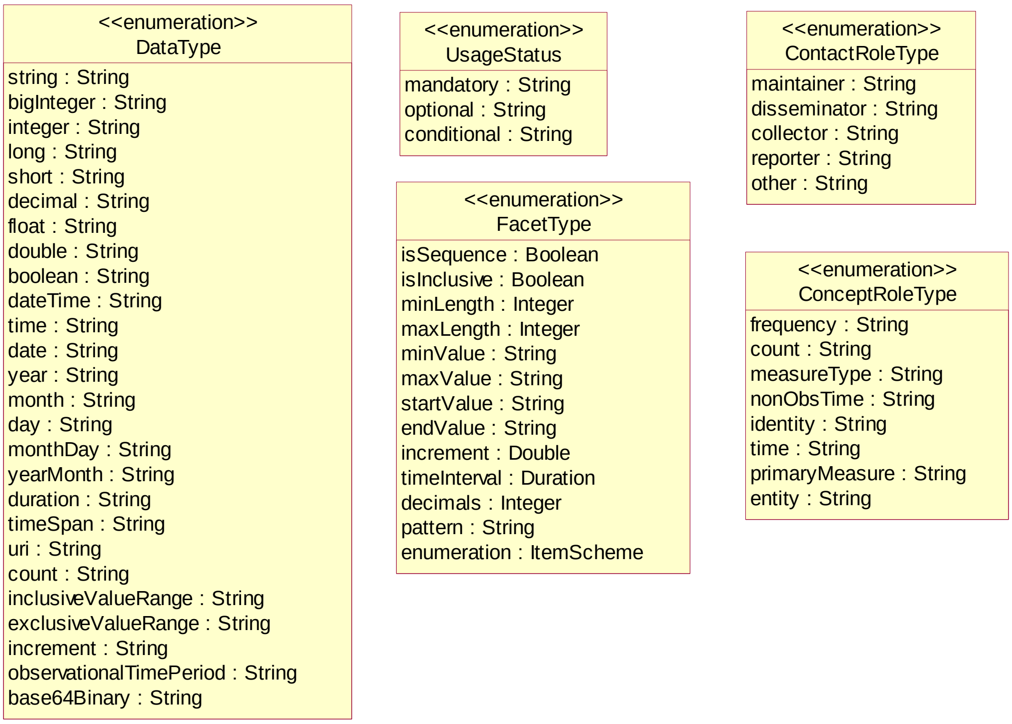
Class	Feature	Description
AnnotableArtefact	Direct sub classes are: *IdentifiableArtefact*	Objects of classes derived from this can have attached annotations.
Annotation		Additional descriptive information attached to an object.
	name	A name used to identify an annotation.
	type	Specifies how the annotation is to be processed.
	url	A link to external descriptive text.
	+text	An International String provides the multilingual text content of the annotation via this role.
IdentifiableArtefact	Superclass is *AnnotableArtefact* Direct sub classes are: *VersionableArtefact*	Provides identity to all derived classes. It also provides annotations to derived classes because it is a subclass of Annotable Artefact.
	id	The unique identifier of the object.

Class	**Feature**	**Description**
	uri	Universal resource identifier that may or may not be resolvable.
	urn	Universal resource name – this is for use in registries: all registered objects have a urn.
	+description	A multi-lingual description is provided by this role via the International String class.
	+name	A multi-lingual name is provided by this role via the International String class
VersionableArtefact	Superclass is *IdentifiableArtefact* Direct sub classes are: *MaintainableArtefact*	Provides versioning information for all derived objects.
	version	A version string following an agreed convention
	validFrom	Date from which the version is valid
	validTo	Date from which version is superceded
InternationalString		The International String is a collection of Localised Strings and supports the representation of a description in multiple locales.
LocalisedString		The Localised String supports the representation of a description in one locale (locale is similar to language but includes geographic variations such as Canadian French, US English etc.).
	label	Label of the string.
	locale	The geographic locale of the string e.g French, Canadian French.
MaintainableArtefact	Inherits from VersionableArtefact Derived classes: *StructureUsage* *Structure* *ItemScheme*	An abstract class to group together primary structural metadata artefacts that are maintained by a MaintenanceAgency.

Class	Feature	Description
	`final`	Defines whether a maintained artefact is draft or final.
	`+maintainer`	Derived classes will be maintained by the MaintenanceAgency specified by this role.
`MaintenanceAgency`		See section on "Organisations"

3.3 Data Types

3.3.1 Class Diagram



3.3.2 Explanation of the Diagram

3.3.2.1 Narrative

The `UsageStatus` enumeration is used as a data type on an attribute where the value of the attribute in an instance of the class must take one of the values in the `UsageStatus` (i.e. mandatory, optional, or conditional).

The `AttributeValueType` enumeration is used as a data type on an attribute value to indicate its format.

The `ConceptRoleType` enumeration is used as a data type on a role attribute to indicate the role that a component plays in a key family (data structure definition). This role is in addition to any formal structural layering of the model such as `Dimension`, *`Measure`*, and `DataAttibute`. The description of the various roles can be found in the section on KeyFamily (section 5).

The `DataType` enumeration is used to specify the valid format of the content of a `Concept` when specified for use on a *`Component`* on a *`Structure`* (such as a `Dimension` in a `KeyFamily`). The description of the various types can be found in the section on Concept Scheme (section 4.4).

The `FacetType` enumeration is used to give context to a specific `facetValue`. The use of this and the description of the various types can be found in the section on Concept Scheme (section 4.4).

3.3.2.2 Definitions

Class	Feature	Description
`UsageStatus`		Lists the possible values that an attribute can take when it is assigned the data type of Usage Status.
	`mandatory`	The usage is mandatory.
	`optional`	The usage is optional.
	`conditional`	The usage is mandatory when certain conditions are satisfied.
`ConceptRoleType`		Lists the possible formats that an attribute value can take when it is assigned as a data type for the attribute (e.g. in Concept Role). The semantic meaning of the role types in the enumeration are defined with the structure in which they are used (e.g. Key Family).
`DataType`		Lists the possible formats that an attribute value can take when it is assigned as a data type for the attribute (e.g. type). The semantic meaning of the data types in the enumeration are defined with the structure in which they are used (e.g. Concept Scheme).

Class	Feature	Description
`FacetType`		Lists the possible formats that an attribute value can take when it is assigned as a data type for the attribute (e.g. facetType). The semantic meaning of the data types in the enumeration are defined with the structure in which they are used (e.g. Concept Scheme).

3.4 The Item Scheme Pattern

3.4.1 Context

The Item Scheme is a basic architectural pattern that allows the creation of list schemes for use in simple taxonomies, for example.

The `ItemScheme` is the basis for `CategoryScheme`, `CodeList`, `ConceptScheme`, and `CodeSet`.

3.4.2 Class Diagram

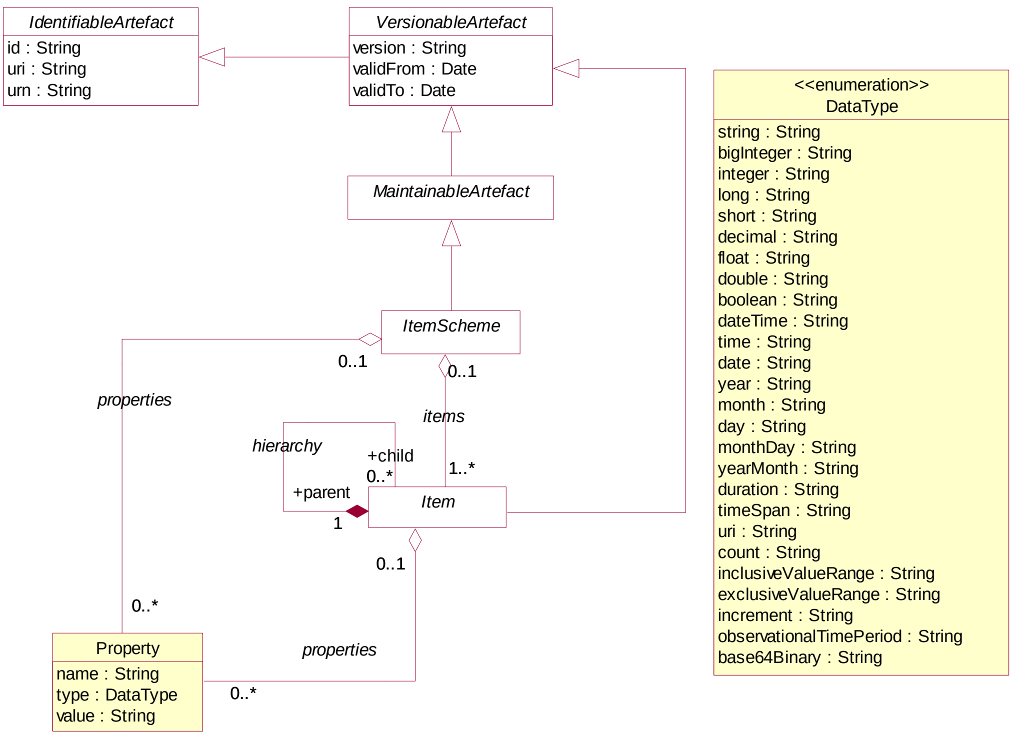


Figure 10 The Item Scheme pattern

3.4.3 Explanation of the Diagram

3.4.3.1 Narrative

The *ItemScheme* is an abstract class which defines a set of *Item* (this class is also abstract). Its main purpose is to define a mechanism which can be used to create taxonomies which can classify other parts of the SDMX Information Model. It is derived from *MaintainableArtefact* which gives it the ability to be annotated, have identity, versioning and be associated with a MaintenanceAgency. An example of concrete classes are CategoryScheme and associated Category.

Item inherits from *VerionableArtefact* which gives it the ability to be annotated, have identity, versioning, and therefore has id, uri and urn attributes, a name and a description in the form of an InternationalString. Unlike the parent *ItemScheme*, and *Item* itself is not a *MaintainableArtefact* and therefore cannot have an independent MaintenanceAgency (i.e. it implicitly has the same agency as the *ItemScheme*).

The *Item* can be hierarchic and so one *Item* can have child *Item*s. The restriction of the hierarchic association is that a child *Item* can have only parent *Item*.

The *ItemScheme*, and the *Item,* can all have optional Property which gives the ability to add extensible properties. The explanation of the various DataTypes can be found in the section on Concept Scheme (section 4.4).

3.4.3.2 Definitions

Class	Feature	Description
ItemScheme	Inherits from: *MaintainableArtefact* Direct sub classes are: CategoryScheme ConceptScheme CodeList OrganisationScheme ItemSchemeAssociation	The descriptive information for an arrangement or division of objects into groups based on characteristics, which the objects have in common.
	property	Association to an Item Property.
Item	Inherits from: *IdentifiableArtefact* Direct sub classes are Category Concept Code Association	The Item is an item of content in an Item Scheme. This may be a node in a taxonomy or ontology, a code in a code list etc.
	hierarchy	This allows an Item optionally to have one or more child Items.
	property	Association to an Item Property.

Class	Feature	Description
`Property`		The specification of a value whose semantic is identified by its name.
	name	The name of the property.
	type	Specifies the data type for the Attribute Property. The types are an enumerated list in the Data Type enumeration..
	value	The value of the property.

3.5 The Structure Pattern

3.5.1 Context

The Structure is a basic architectural pattern which allows the specification of complex tabular structures which are often found in statistical data (such as key family, cube, and metadata structure definitions). A Structure is a set of ordered lists. A pattern to underpin this tabular structure has been developed, so that commonalities between these structure definitions can be supported by common software and common syntax structures.

3.5.2 Class Diagram

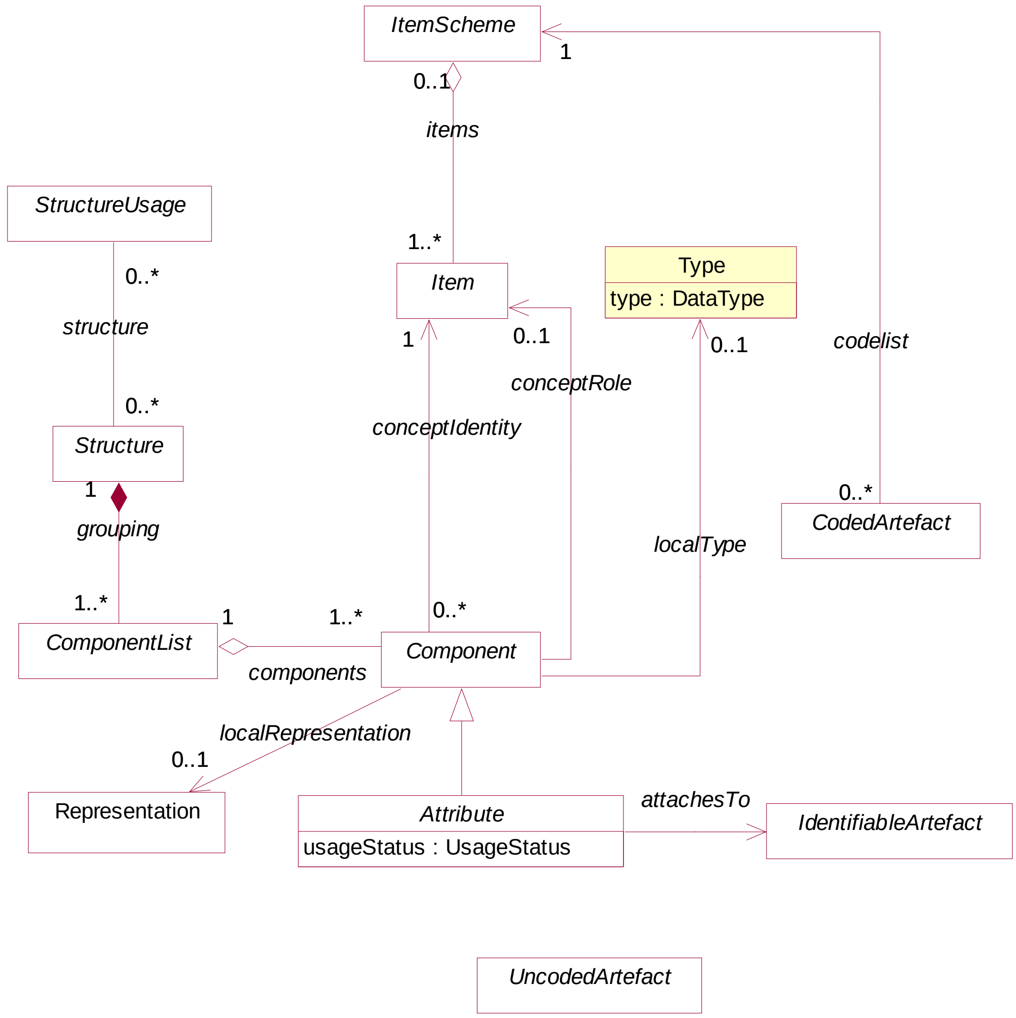


Figure 11 The Structure pattern

3.5.3 Explanation of the Diagram

3.5.3.1 Narrative

The *`Structure`* is an abstract class which contains a set of one or more *`ComponentList`*(s) (this class is also abstract). An example of a concrete *`ComponentStructure`* is `KeyFamily`. The *`ComponentList`*(s) are embedded within the *`Structure`*, and this is indicated by the solid diamond on the grouping association.

The *`ComponentList`* is a list of one or more *`Component(s)`*. The *`ComponentList`* has several concrete descriptor classes based on it: `KeyDescriptor`, `GroupKeyDescriptor`, `MeasureDescriptor`, and `AttributeDescriptor` of the KeyFamily are examples. In the case of a KeyDescriptor acting as a *ComponentList*, its *Component*(s) would be Dimension(s).

Each Component takes its semantic (and possibly also its representation) from an Item in an *ItemScheme*, such as a Concept in a ConceptScheme. Furthermore, a *Component* may be defined as having one or more roles in the structure, and this is identified by the +conceptRole association to an *Item* in an *ItemScheme* that defines roles. The *Component* may also have a Type specified localType, this allows a concrete class, such as Dimension, to specify a data type that is local to the *Structure* in which it is contained (for Dimension this will be KeyFamily), and thus overrides any Type specified for the Item which contains its conceptIdentity (in the case of a Dimension this would be a Concept).

A specific sub class of Component is the Attribute. Attributes are used in specific Structures (such as a KeyFamily) and are specified as being "attachable" to specific components in the model. This is supported by the association "attachesTo" which links to an *IdentifiableArtefact*. This association is constrained in the concrete models that use this structure pattern in order to specify the actual model components to which the attribute can be attached.

The *Structure* may be used by one or more *StructureUsage*. An example of this in terms of concrete classes is that a DataflowDefinition (sub class of *StructureUsage*) may use a particular KeyFamily (sub class of *Structure*), and similar constructs apply for the MetadataflowDefinition (link to MetadataStructureDefinition) and the CubeDefinition (link to CubeStructure).

Finally, the pattern contains *CodedArtefact* and *UncodedArtefact*. The model distinguishes between two fundamental "representations" for components in a structure. The *CodedArtefact* associates an *ItemScheme* (usually a CodeList) that defines its valid content, whilst an *UncodedArtefact* does not have a link to a formal list that specifies valid content. However, an *UncodedArtefact* may have a specific non coded representation other than text. The valid representations are described in the section 4.4 (Concept Scheme).

547 **3.5.3.2 Definitions**

Class	Feature	Description
StructureUsage	Inherits from: *MaintainableArtefact* Direct sub classes are: DataflowDefinition (see Figure 22) MetadataflowDefinition (see Figure 22)	An artefact whose components are described by a Structure. In concrete terms (sub-classes) an example would be a Dataflow Definition which is linked to a given structure – in this case the Key Family.
	structure	An association to a Structure specifying the structure of the artefact.
Structure	Inherits from: *MaintainableArtefact* Direct sub classes are: KeyFamily MetadataStructure Definition	Abstract specification of a list of lists to define a complex tabular structure. A concrete example of this would be statistical concepts, code lists, and their organisation in a data or metadata structure definition, defined by a centre institution, usually for the exchange of statistical information with its partners.
	grouping	A composite association to one or more component lists.
ComponentList	Inherits from: *IdentifiableArtefact* Direct sub classes are: KeyDescriptor GroupKeyDescriptor MeasureDescriptor AttributeDescriptor TargetIdentifier PartialTarget Identifier ConceptDescriptor	An abstract definition of a list of components. A concrete example is a key descriptor which defines the list of dimensions that make up a key for a key family.
	components	An aggregate association to one or more components which make up the list.

Class	Feature	Description
Component	Inherits from: IdentifiableArtefact Direct sub classes are: *Measure* *Attribute* Dimension IdentifierComponent	A component is an abstract super class used to define qualitative and quantitative data and metadata items that belong to a Component List and hence a Structure. Component is refined through its sub-classes.
Attribute	Inherits from: *Component* Direct sub classes are: UncodedDataAttribute CodedDataAttribute *MetadataAttribute*	An abstract class used to provide qualitative information.
	usageStatus	Defines the usage status which is constrained by the data type Usage Status.
UncodedArtefact	Direct sub classes are: UncodedDataAttribute UncodedMetadata Attribute UncodedMeasure	An uncoded artefact is an abstract class used to define qualitative, quantitative or free text values which are not drawn from a maintained value set.
CodedArtefact	Direct sub classes are: Dimension CodedDataAttribute CodedMeasure IdentifierComponent CodedMetadata Attribute	A coded artefact is an abstract class used to define qualitative values which are drawn from a maintained value set.
	codelist	An association to an Item Scheme which allows sub-classes to define the code list from which this component takes its values.

3.6 Association Pattern

3.6.1 Context

The Structure is a basic architectural pattern which allows the specification of complex tabular structures which are often found in statistical data (such as key family,

3.6.2 Class Diagram

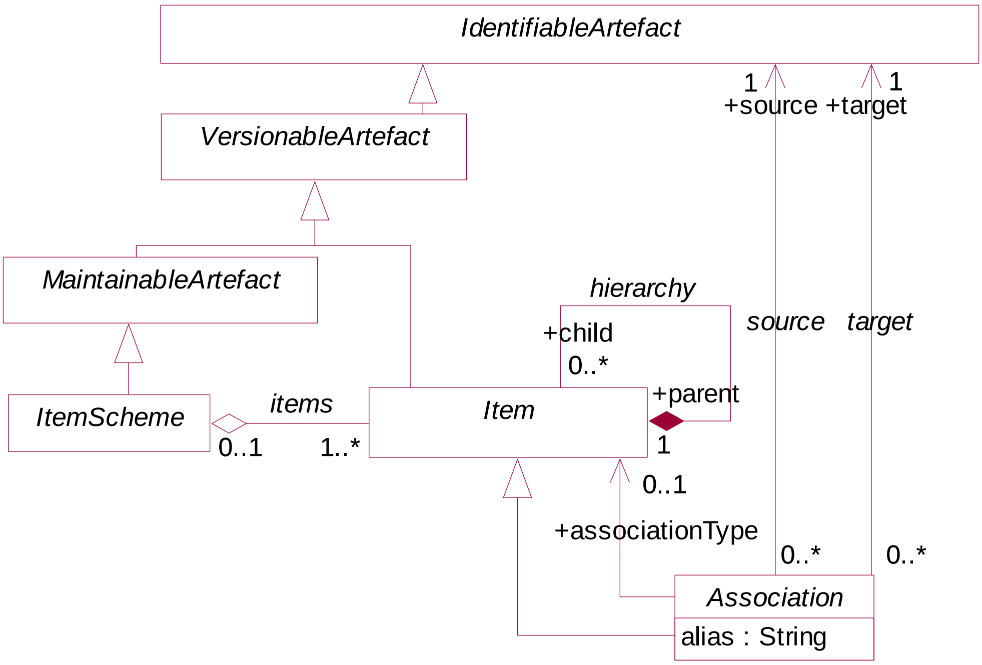


Figure 12: Class diagram of the Association Pattern

3.6.3 Explanation of the Diagram

3.6.3.1 Narrative

The Association Pattern permits associations between any two *`IdentifiableArtefact`*s. The association has a coded type specified by an Item in an *`ItemScheme`*. The Association is a *`VersionableArtefact`*, allowing associations between objects to evolve over time. The *`Association`* is also an *`Item`*, thus it can contain child *`Association`*s. This is useful for expressing mapping between lists and hierarchies. For example, an Association may map two `CodeList`s together and a set of children *`Association`*s would map the individual `Code`s. A more elaborate hierarchy would be to map all components in a KeyFamily, including the `CodeLists` and `Codes` used by the components. Schematically this would be:

`KeyFamily`➔`[Dimension, DataAttribute, Measure]`➔`CodeList`➔`Code`.

The specific use of this pattern is described in Structure Set (section 9).

The `alias` attribute is used to specify a neutral name which can refer to multiple pair-wise mappings thus facilitating querying across a set of mapped artefacts.

3.6.3.2 Definitions

Class	Feature	Description
`Association`	Inherits from *`Item`*	Links two Identifiable Artefacts in a source and target association.

Class	Feature	Description
	`+source`	Association to the source Identifiable Artefact.
	`+target`	Association to the target Identifiable Artefact.
	`+associationType`	Association to an Item that specifies the role of the link between the source and target Identifiable Artefact.
	`alias`	Specifies a neutral name which can refer to multiple pair-wise mappings of Identifiable Artefacts.

3.7 Inheritance

3.7.1 Class Diagram

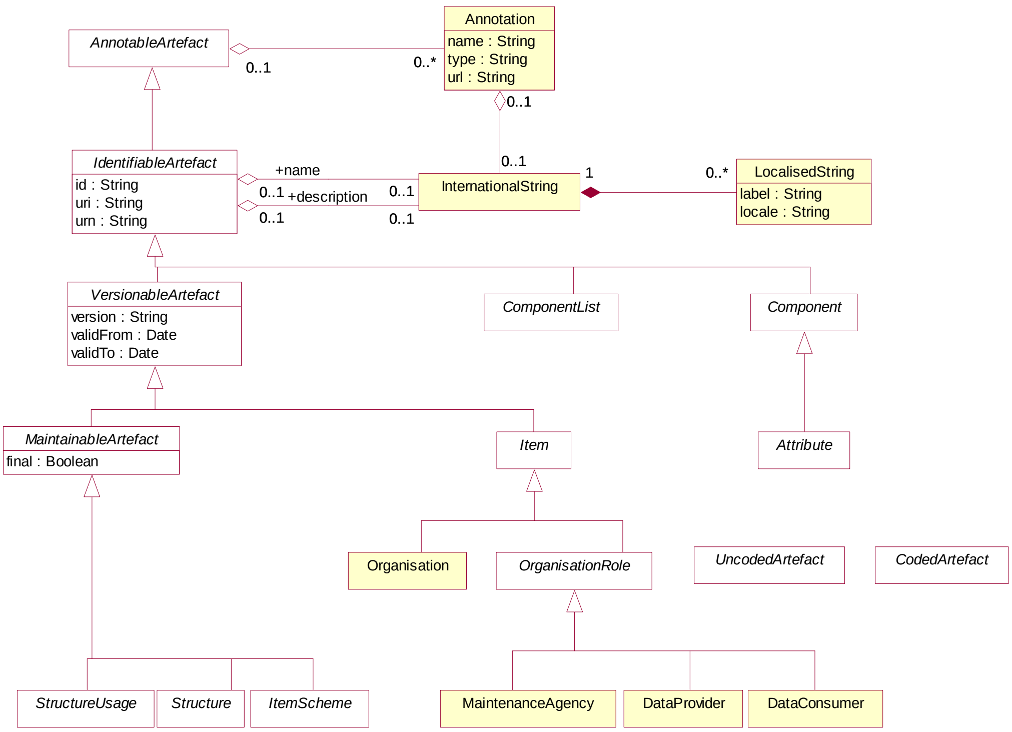


Figure 13 Inheritance within the base structures

3.7.2 Explanation of the Diagram

3.7.2.1 Narrative

The diagram above shows the inheritance within the base structures. Many of the concrete classes are introduced and defined in the specific package to which they relate: principally the Data Structure Definition and the Metadata Structure Definition.

Note that neither *`CodedArtefact`* nor *`UncodedArtefact`* inherit from any of the base classes and in themselves they have no identification. It will be seen later that the concrete classes that inherit from these classes also inherit from a class which does have identification (e.g. in the case of a data attribute this is `CodedDataAttribute`).

4 SPECIFIC ITEM SCHEMES

4.1 Introduction

The structures that are an arrangement of objects into hierarchies or lists based on characteristics, and which are maintained as a group inherit from *`ItemScheme`*. These concrete classes are:

- `CodeList`
- `ConceptScheme`
- `CategoryScheme`
- `ObjectTypeScheme`
- `OrganisationScheme`
- `ItemSchemeAssociation`
- `TransformationScheme`

The `TransformationScheme` is described in the section on Transformations and Expressions (section 12). This section describes the remaining specialisations of the *`ItemScheme`*.

4.2 Inheritance View

The inheritance and relationship views are shown together in each of the diagrams below.

4.3 Code List

4.3.1 Class Diagram

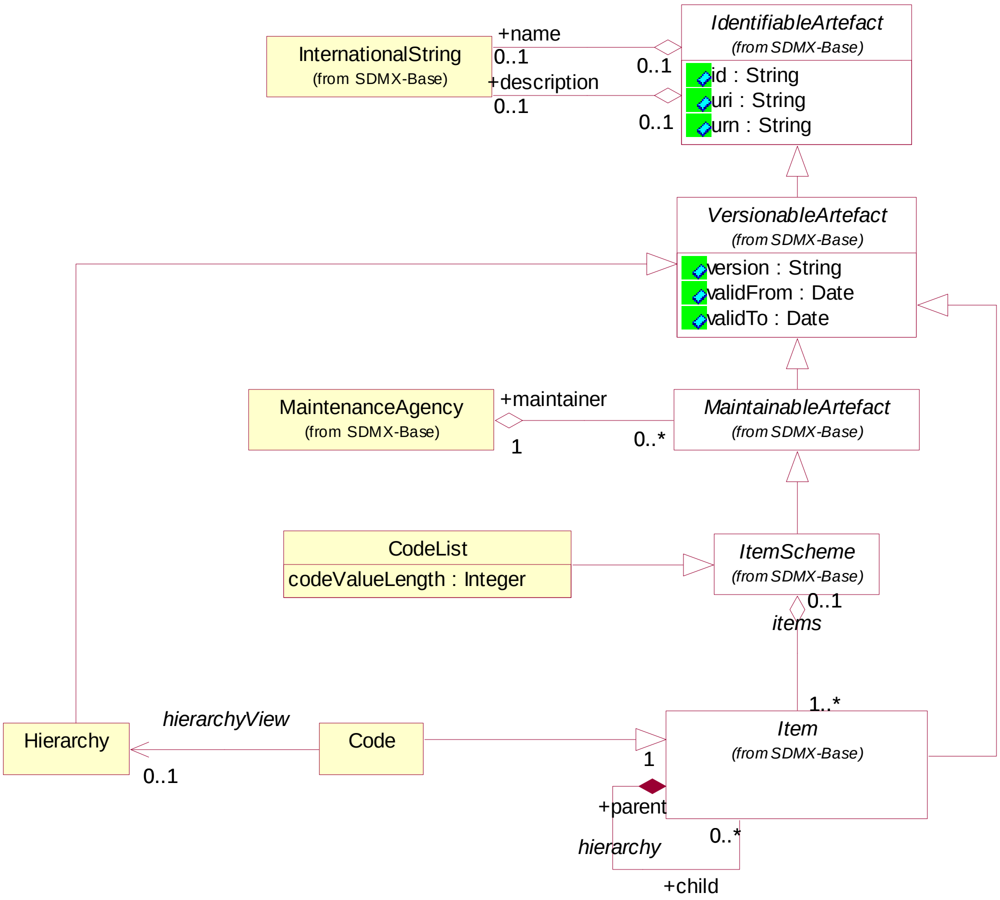


Figure 14 Class diagram of the Code List

4.3.2 Explanation of the Diagram

4.3.2.1 Narrative

The `CodeList` inherits from the *`ItemScheme`* and the `Code` inherits from the *`Item`* and both therefore have the following attributes:

- `id`
- `uri`
- `urn`
- `version`

- `validFrom`
- `validTo`

They also have the association to `InternationalString` to support a multi-lingual name, an optional multi-lingual description, and an association to `Annotation` to support notes (not shown).

Through the inheritance the `CodeList` comprise one or more `Codes`, and the `Code` itself can have one or more child `Codes` in the `hierarchy` association . Note that a child `Code` can have only one parent `Code` in this association. A more complex `CodeSet` which allow multiple parents and multiple hierarchies is described later. A more complex `HierachicalCodeScheme` which allow multiple parents and multiple hierarchies is described later. In the `HierachicalCodeScheme` the `Code` is referenced from the `HierarchicalCodeScheme`, but there may be a requirement to link from the `Code` to the *`Hierarchy`* in a `HierarchicalCodeScheme` (such a link will support code mappings – see section 9).and this is supported via the `hierarchyView` association.

4.3.2.2 Definitions

Class	**Feature**	**Description**
`CodeList`	Inherits from *`ItemScheme`*	A list from which some statistical concepts (coded concepts) take their values. In this model the coded concepts are the sub classes of the Coded Artefact.
	`codeValueLength`	The length of a code (i.e. identifier) in the code list.
	`/items`	Associates the codes.
	`/`	
`Code`	Inherits from *`Item`*	A language independent set of letters, numbers or symbols that represent a concept whose meaning is described in a natural language.
	`hierarchy`	Associates the parent and the child codes.
	`hierachyView`	Associates a Hierarchy

4.4 Concept Scheme

4.4.1 Inheritance Class Diagram

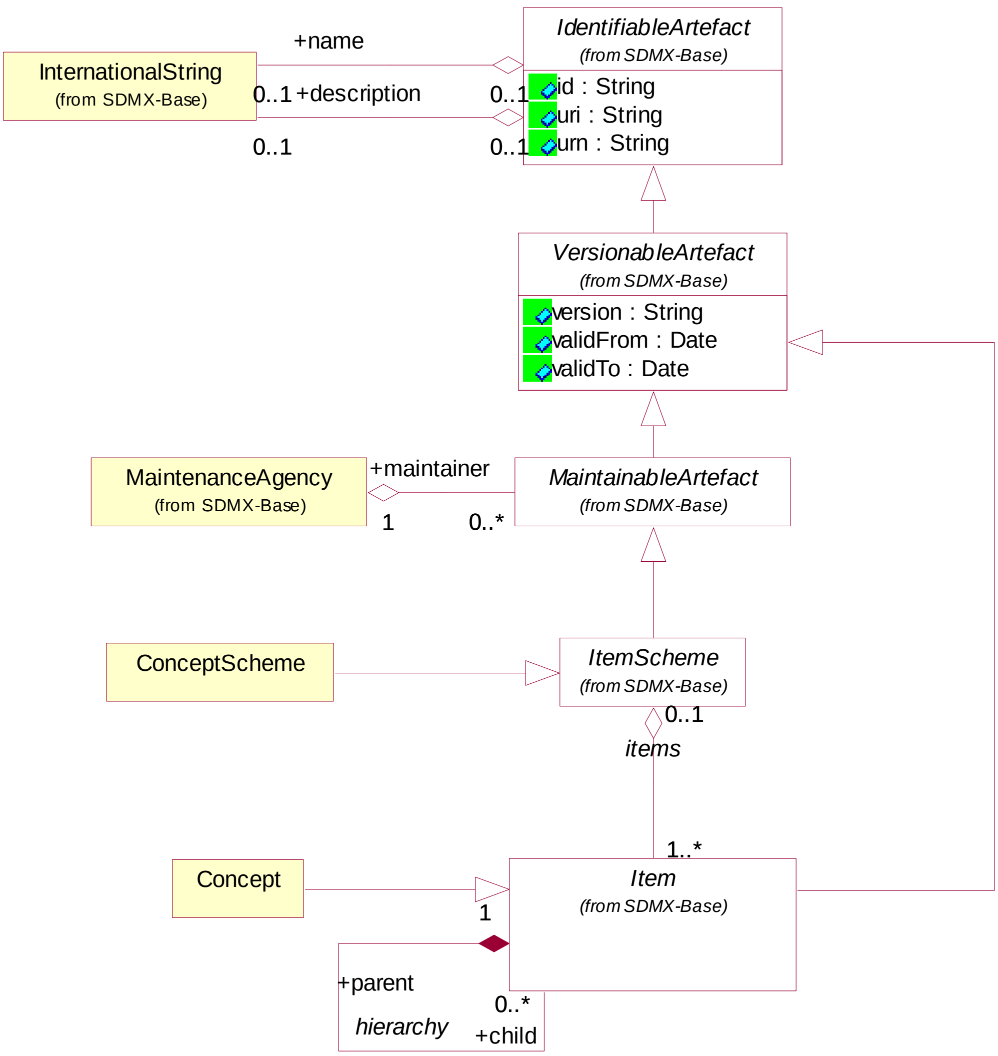


Figure 15 Class diagram of the Concept Scheme

4.4.2 Explanation of the Diagram

The `ConceptScheme` inherits from the *`ItemScheme`* and the `Concept` inherits from the *`Item`*, and therefore both have the following attributes:

- `id`
- `uri`
- `urn`

- `version`
- `validFrom`
- `validTo`

Both also have the association to `InternationalString` to support a multi-lingual name, an optional multi-lingual description, and an association to `Annotation` to support notes (not shown).

4.4.3 Relationship class Diagram

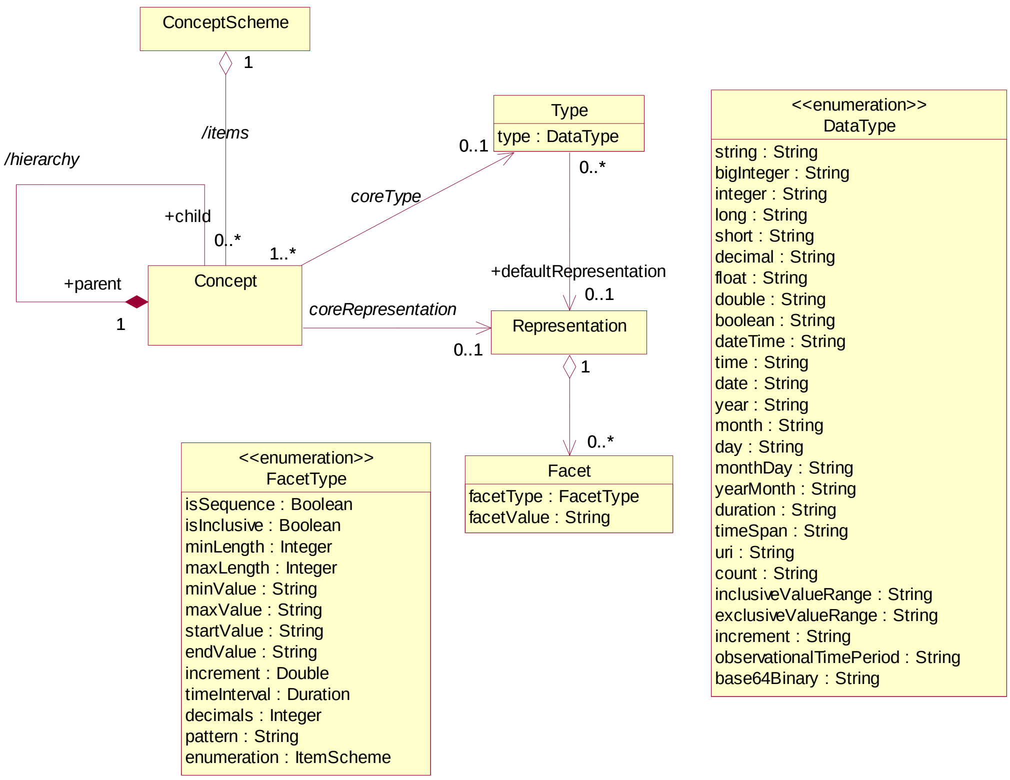


Figure 16: Relationship class diagram of the Concept Scheme

4.4.4 Explanation of the diagram

4.4.4.1 Narrative

The `ConceptScheme` can have one or more `Concept`. A `Concept` can have zero or more child `Concept`, thus supporting a hierarchy of `Concepts`. Note that a child `Concept` can have only one parent `Concept` in this association. The purpose of the hierarchy is to relate concepts that have a semantic relationship: for example a Reporting_Country and Vis_a_Vis_Country may both have Country as a parent concept, or a CONTACT may have a PRIMARY_CONTACT as a child concept. It is not the purpose of such schemes to define reporting structures: these reporting structures are defined in the `KeyFamily` or the `MetadataStructureDefinition`.

The `Concept` can be defined as conforming to a specified `Type` such as string, numeric etc. which is its `coreType` and it may also have a specified

`Representation` which is the `coreRepresentation` i.e. the `coreType` and `coreRepresentation` is the specification of the format and value domain of the `Concept` when used on a structure like a `KeyFamily` or a `MetadataStructureDefinition` unless the specification of the `Type` or `Representation` is overridden in the relevant structure definition. In a hierarchical `ConceptScheme` the `Type` and `Representation` are inherited from the parent `Concept` unless overridden at the level of the child `Concept`.

Note that whilst the `Representation` is dependent upon the value of the `Type.DataType` (this is the association with the role `defaultRepresentation`) this is not shown as mandatory on the model, for reasons of compatibility with version 1.0, which does not support all the *`Representation`*s.

Note that whilst the `Representation` is dependent upon the value of the `Type.DataType` (this is the association with the role `defaultRepresentation`) this is not shown as mandatory on the model, for reasons of compatibility with version 1.0, which does not support all the *`Representation`*s.

The majority of SDMX data types are compatible with those found in XML Schema, and have equivalents in most current implementation platforms:

SDMX Data Type	XML Schema Data Type	.NET Framework Type	Java Data Type
String	xsd:string	System.String	java.lang.String
BigInteger	xsd:integer	System.Decimal	java.math.BigInteger
Integer	xsd:int	System.Int32	int
Long	xsd.long	System.Int64	long
Short	xsd:short	System.Int16	short
Decimal	xsd:decimal	System.Decimal	java.math.BigDecimal
Float	xsd:float	System.Single	float
Double	xsd:double	System.Double	double
Boolean	xsd:boolean	System.Boolean	boolean
DateTime	xsd:dateTime	System.DateTime	javax.xml.datatype.XMLGregorianCalendar
Time	xsd:time	System.DateTime	javax.xml.datatype.XMLGregorianCalendar
Date	xsd:date	System.DateTime	javax.xml.datatype.XMLGregorianCalendar
Year, Month, Day, MonthDay, YearMonth	xsd:g*	System.DateTime	javax.xml.datatype.XMLGregorianCalendar
Duration	xsd:duration	System.TimeSpan	javax.xml.datatype.Duration
Base64Binary	xsd:base64Binary	System.Byte[]	byte[]
URI	xsd:anyURI	System.Uri	Java.net.URI or java.lang.String

There are also a number of SDMX data types which do not have these direct correspondences, often because they are composite representations:

- Timespan (start DateTime + Duration)
- ObservationalTimePeriod (a union type of Date, Time, DateTime, and a set of codes for common periods – see Implementor's Guide).

As stated previously, the value domain of a `Type` is expressed by a `Representation`. The `Representation` is composed of `Facets`, each of which conveys characteristic information related to the definition of a value domain. Often a set of `Facet(s)` are needed to convey the required semantic. For example, a sequence is defined by a minimum of two Facets: one to define the start value, and one to define the increment. Semantically legal combinations of `Facets` depend upon the Type that they restrict, but are selected from the following table of `facetTypes`.

Facet Type	**Explanation**
isSequence	If true, the Representation is an incremental sequence of integer values (value range) or date/time values (time range). The facets startValue, and interval or timeInterval must also be specified for a sequence.
isInclusive	If true, valid values for the Representation lie within the given value/time range, otherwise outside the value/time range.
minLength	Specifies the minimum number of characters for a value.
maxLength	Specifies the maximum number of characters for a value.
minValue	Specifies the minimum numeric value.
maxValue	Specifies the maximum numeric value.
startValue	Specifies the starting value for a sequence (time or value range).
endValue	Specifies the end value for a sequence (time or value range).
increment	Used to specify the incremental steps of a value range. Starting from startValue, and incrementing by increment until endValue is reached. The sequence then begins again from startValue. If no endValue is specified, the sequence continues indefinitely.
timeInterval	Used to specify the incremental steps (periods) of a time range. Starting from startValue, and incrementing by timeInterval until endValue is reached. The sequence then begins again from startValue. If no endValue is specified, the sequence continues indefinitely.
decimals	The Representation has a specified number of decimals.
pattern	The Representation is a regular expression (see XSD spec) which is expressed as a string.
enumeration	The Representation is an enumeration of Items in specific scheme of Items, such as an identified Code List.

4.4.4.2 Definitions

Class	Feature	Description
ConceptScheme	Inherits from *ItemScheme*	The descriptive information for an arrangement or division of concepts into groups based on characteristics, which the objects have in common.
	/items	Associates the concept.
Concept	Inherits from *Item*	A concept is a unit of knowledge created by a unique combination of characteristics.
	/hierarchy	Associates the parent and the child concept.
	coreType	Associates a data Type.
	coreRepresentation	Associates a Representation.
Type	type	Specifies, as a mnemonic, the valid format of the content that can be reported such as Alpha, Num, Time.
Representation	Abstract class Sub classes: ItemScheme DataRange NumericRange Pattern	Specifies the content of the Concept when reported in a Data Set or a Metadata Set.
DateRange		A data range and periodicity of the dates in the range.
	startDate	The start date of the date range.
	endDate	The end date of the date range.
	periodicity	The time periodicity by which a set of dates can be implied by incrementing by the periodicity from the start date up to the end date.
NumericRange		A numeric range and the increment of the numbers

Class	Feature	Description
		in the range.
	`maxValue`	The maximum value in the range.
	`minValue`	The minimum value in the range
	`increment`	The increment by which a set of values can be implied by incrementing from the start or minimum value.
`Pattern`		A representation that is in the form of a pattern that can be expressed as an expression.
	`regularExpression`	An expression that defines the format of data or metadata content.
`Sequence`		A sequence of whole numbers.
	`startValue`	The start value in a sequence of values
	`increment`	The increment by which a set of values can be implied by incrementing from the start or minimum value.

4.5 Category Scheme

4.5.1 Context

This package defines the structure that supports the definition of and relationships between categories in a category scheme. It is similar to the package for concept scheme. An example of a category scheme is one which categorises data – sometimes known as a subject matter domain scheme or a data category scheme. Another example is a reporting taxonomy scheme which defines the conceptual structure of a reporting scheme which has, at its leaves, many individual “sets” of data each described by a specific structure definition (this is the type of report that is typically found in primary reporting). Importantly, as will be seen later, the individual nodes in the scheme (the “categories”) can be associated to actual dataflows which in turn links to the definition of the structure of the dataflow (i.e. `KeyFamily`).

4.5.2 Class diagram

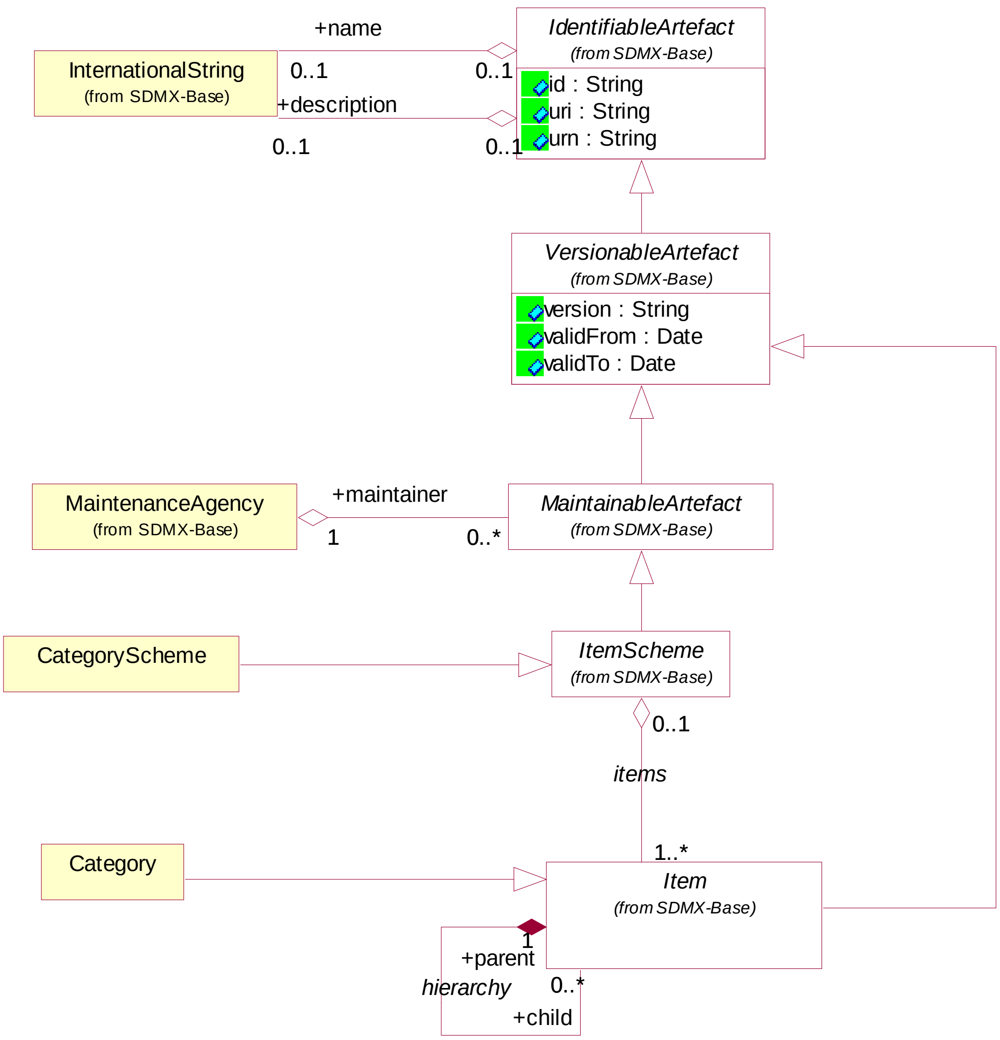


Figure 17 Class diagram of the Category Scheme

4.5.3 Explanation of the Diagram

4.5.3.1 Narrative

The categories are modelled as a hierarchical *`ItemScheme`*. The `CategoryScheme` inherits from the *`ItemScheme`* and the `Category` inherits from the *`Item`*, and therefore both have the following attributes:

- `id`
- `uri`
- `urn`

- version
- validFrom
- validTo

Both also have the association to InternationalString to support a multi-lingual name, an optional multi-lingual description, and an association to Annotation to support notes (not shown on the model).

The CategoryScheme can have one or more Category. A Category can have zero or more child Category, thus supporting a hierarchy of Categorys. Note that a child Category can have only one parent Category in this association. A more complex CodeSet which allow multiple parents and multiple hierarchies is modelled later.

4.5.3.2 Definitions

Class	Feature	Description
CategoryScheme	Inherits from *ItemScheme*	The descriptive information for an arrangement or division of categories into groups based on characteristics, which the objects have in common.
	/items	Associates the category.
Category	Inherits from *Item*	An item at any level within a classification, typically tabulation categories, sections, subsections, divisions, subdivisions, groups, subgroups, classes and subclasses.
	hierarchy	Associates the parent and the child Category.

4.6 Object Type Scheme

4.6.1 Context

It may be necessary in an SDMX document to identify an object type that is in the SDMX model. An example of such a document is a Metadata Structure Definition which specifies the attachment of metadata to a Dataflow, or a Key Family, or a Code List etc. It is necessary in such a definition to identify the object type and this must be taken from a valid “list” of object types. The ObjectTypeScheme is such a list.

4.6.2 Class Diagram

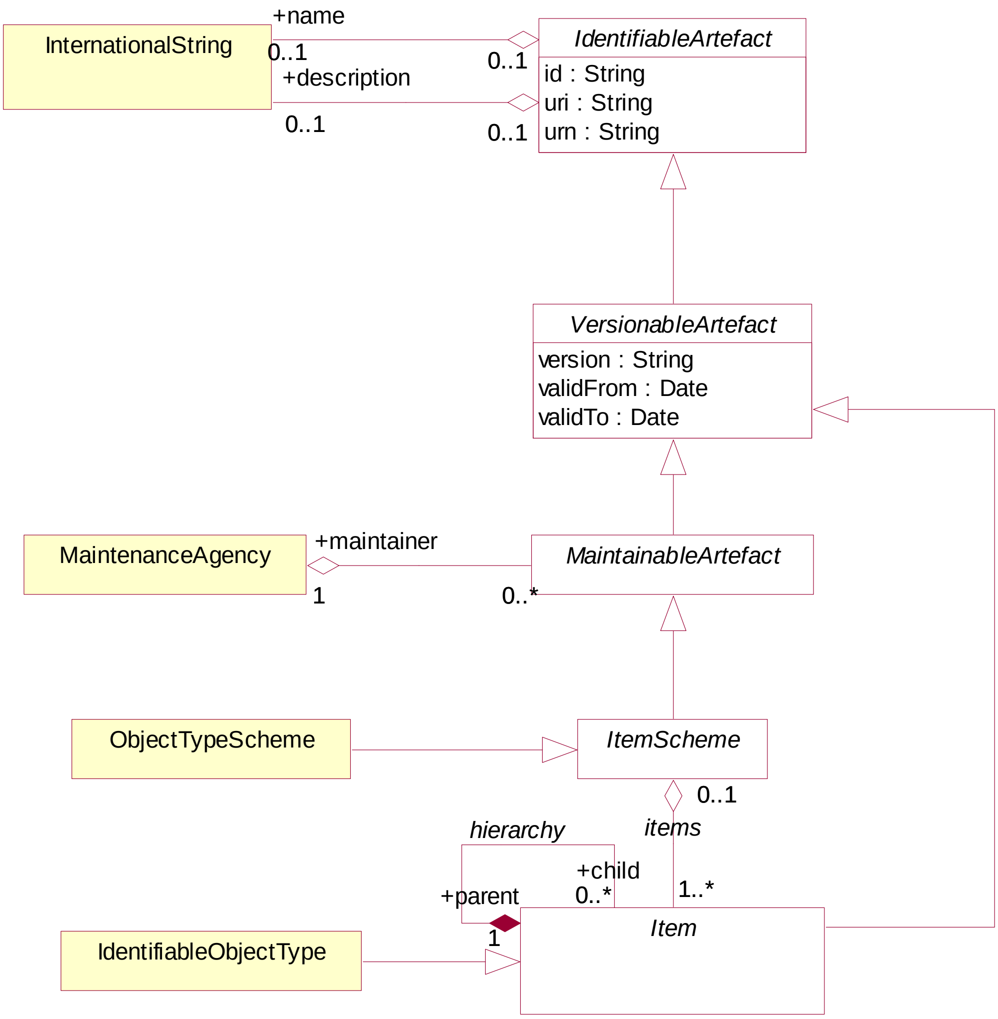


Figure 18: Class diagram of the Object Type Scheme

4.6.3 Explanation of the diagram

4.6.3.1 Narrative

The object types are modelled as an *ItemScheme*. The ObjectTypeScheme inherits from the *ItemScheme* and the IdentifiableObjectType inherits from the *Item*, and therefore both have the following attributes:

- id
- uri
- urn
- version
- validFrom

- `validTo`

Both also have the association to `InternationalString` to support a multi-lingual name, an optional multi-lingual description, and an association to `Annotation` to support notes (not shown on the model).

The `ObjectTypeScheme` can have one or more `IdentifiableObjectType`.

4.6.3.2 Definitions

Class	Feature	Description
`ObjectTypeScheme`	Inherits from *`ItemScheme`*	A collection of identifiable object types (also known as classes or entitities) that may be contained in a data model or other artefact defining or describing object types.
	`/items`	Associates the identifiable object type.
`IdentifiableObject Type`	Inherits from *`Item`*	Description of a set of objects that share the same attributes, operations, methods, relationships, and semantics, and which has identity so that an instance of the object type (i.e. an individual object) may be referenced.

4.7 Type Scheme

4.7.1 Context

This is a scheme of types such as data types. It is used to associate a type with another artefact such an `ExpressionNode` where the type defines the expected data type of the result of the expression defined in the `ExpressionNode`. (see TRANSFORMATIONS AND EXPRESSIONS).

4.7.2 Class Diagram

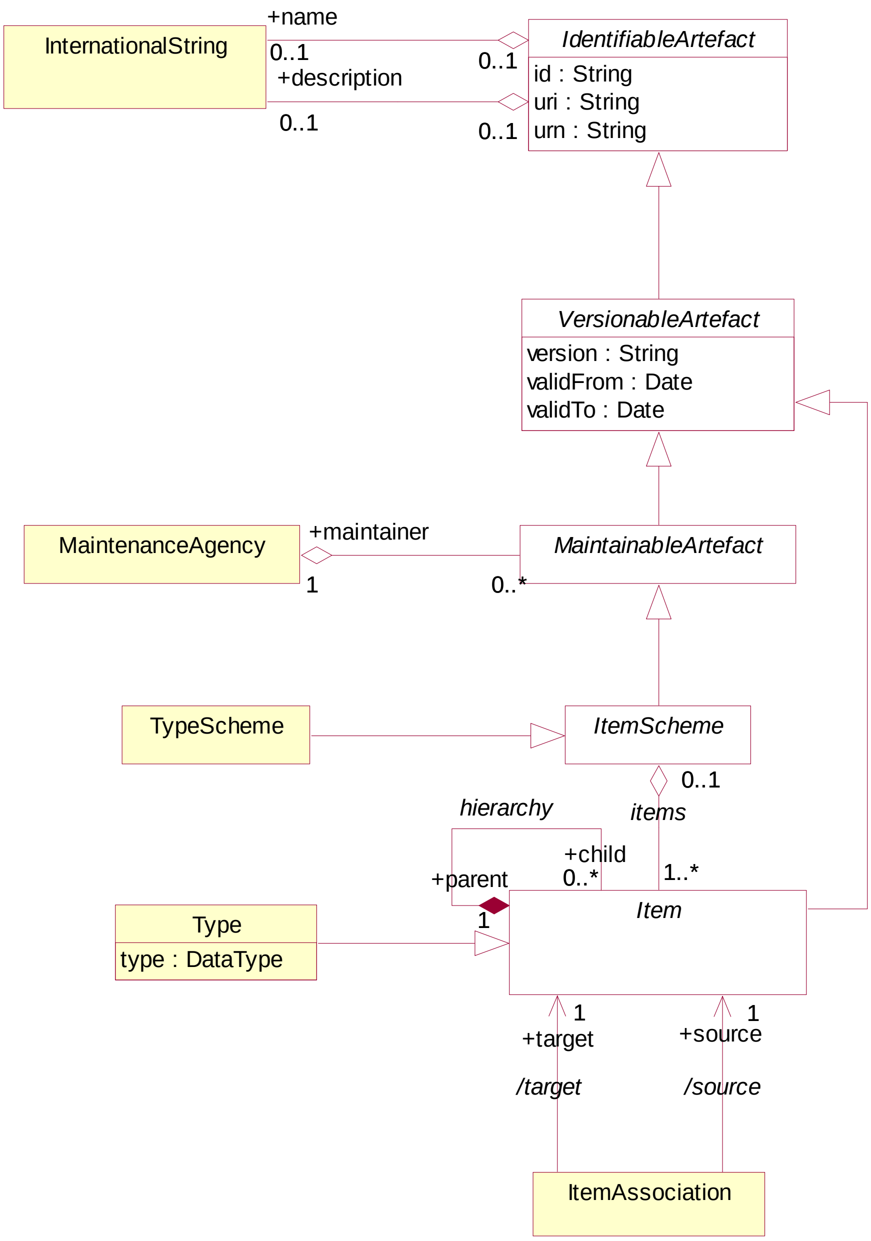


Figure 19: Class diagram of the Type Scheme

4.7.3 Explanation of the Diagram

4.7.3.1 Narrative

The types are modelled as an *`ItemScheme`*. The `TypeScheme` inherits from the *`ItemScheme`* and the `Type` inherits from the *`Item`*, and therefore both have the following attributes:

- `id`

- uri
- urn
- version
- validFrom
- validTo

Both also have the association to InternationalString to support a multi-lingual name, an optional multi-lingual description, and an association to Annotation to support notes (not shown on the model).

The TypeScheme can have one or more Types.

4.7.3.2 Definitions

Class	Feature	Description
TypeScheme	Inherits from *ItemScheme*	A collection of items that define the valid format of data so that such data can be processed by a computer system.
	/items	Association to the Types in the scheme.
Type	Inherits from *Item*	Specifies a data format such that it can be processed accordingly in a computer system, such as numeric or string.
	type	Identification of the type.

4.8 Organisation Scheme

4.8.1 Class Diagram

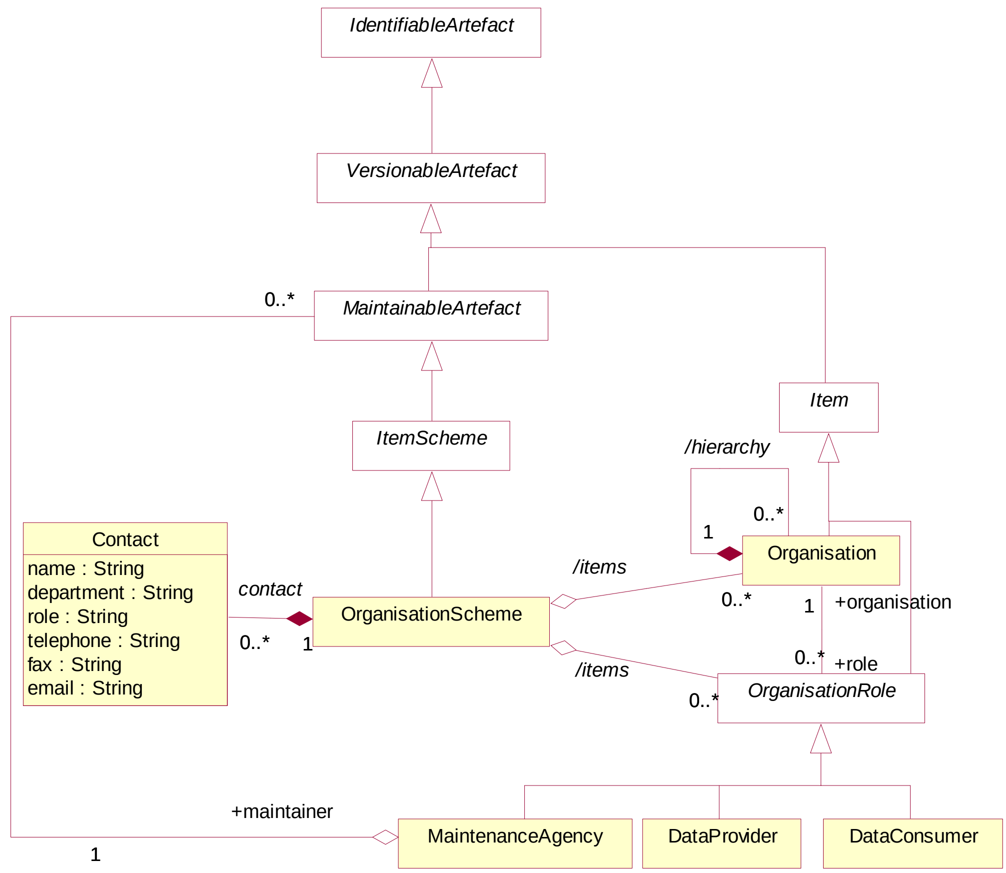


Figure 20 The Organisation class diagram

4.8.2 Explanation of the Diagram

4.8.2.1 Narrative

The `Organisation` inherits from *`Item`* and so has identity and version information, and is maintained in an `OrganisationScheme` (which itself is a sub class of *`ItemScheme`*). An `Organisation` can play a number of *`OrganisationRole`*. Three roles are identified at present: `DataProvider`; `DataConsumer`; `MaintenanceAgency`.. The classes that are associated with these roles are defined in the package(s) where they are relevant. Note that the role `DataProvider` and `DataConsumer` also embrace the activity of metadata provision and consumption.

The model allows the `OrganisationScheme` to be navigated by one or both of Organisation and *`OrganisationRole`*. However, whilst an `Organisation` can play many *`OrganisationRoles`* it is recommended that any one

`OrganisationScheme` contains just one *`OrganisationRole`* (i.e. one of `DataProvider`, `DataConsumer`, or `MaintenanceAgency`).

Metadata can be attached to the *`OrganisationRole`* by means of the metadata attachment mechanism. This mechanism is explained in the Reference Metadata section of this document (see section 7). This means that the model does not specify the specific metadata that can be attached to a `DataProvider` or `MaintenanceAgency`, such as contact information, as this can be provided dynamically using the metadata attachment mechanism.

A limited set of `Contact` information can be attached at the level of the `OrganisationScheme`. If more contact information is required this can be achieved via Reference Metadata.

The `MaintenanceAgency` can maintain a variety of *`MaintainableArtefact`*. The *`MaintainableArtefact`* is an abstract class and the concrete classes are shown at the beginning of the relevant sections in which they are described.

4.8.2.2 Definitions

Class	Feature	Description
`OrganisationScheme`	Inherits from *`ItemScheme`*	A maintained collection of Organisations.
	`contact`	Association to the Contact information foe the scheme.
	`/items`	Association to the Organisations in the scheme.
	`/items`	Association to the Organisation Roles in the scheme.
`Contact`		An instance of a role of an individual or an organization (or organization part or organization person) to whom an information item(s), a material object(s) and/or person(s) can be sent to or from in a specified context.
	`name`	The designation of the Contact person by a linguistic expression.
	`department`	The designation of the organisational structure by a linguistic expression, within which Contact person works.

Class	Feature	Description
	role	The responsibility of the Contact person with respect to the object for which this person is the Contact.
	telephone	The telephone number of the Contact.
	fax	The fax number of the Contact.
	email	The Internet e-mail address of the Contact.
Organisation	Inherits from *Item*	An organisation is a unique framework of authority within which a person or persons act, or are designated to act, towards some purpose.
	/hierarchy	Association between two Organisations in a parent/child relationship.
	+role	Association to the Organisation Role
OrganisationRole	Inherits from *Item*	The function or activities of an organisation, in statistical processes such as collection, processing and dissemination"
	+organisation	Association to the Organisation.
MaintenanceAgency	Inherits from *OrganisationRole*	Responsible agency for maintaining artefacts such as statistical classifications, glossaries, key family structural definitions, and metadata structure definitions.
DataProvider	Inherits from *OrganisationRole*	An organisation that produces data or reference metadata.
DataConsumer	Inherits from *OrganisationRole*	An organisation using data as input for further processing.
MaintainableArtefact		See section on Identification, versioning, and maintenance.
	+Maintainer	An association to the maintenance agency.

4.9 Item Scheme Association

4.9.1 Context

The `ItemSchemeAssociation` is used to associate the *`Item`*s in two different *`ItemSchemes`*. This is a generic mechanism that can be used to map *`Items`*. Specific models exist for mapping schemes where there is a semantic equivalence between *`Item`*s in the *`ItemScheme`*. The models exist for:

- `CodeList`
- `ConceptScheme`
- `CategoryScheme`

These can be found in section 9 - STRUCTURE SET AND MAPPINGS.

4.9.2 Class Diagram

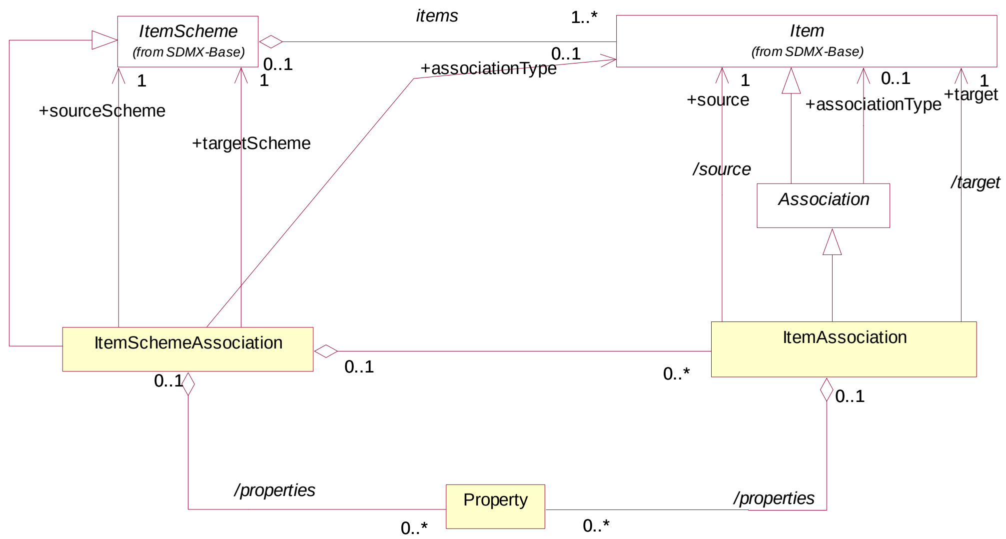


Figure 21: Class diagram of the Item Scheme Association

4.9.3 Explanation of the Diagram

4.9.3.1 Narrative

The `ItemSchemeAssociation` inherits from *`ItemScheme`* and the `ItemAssociation` inherits form Item and therefore both in inherit the ability to have associated Property – thus allowing for the definition of additional metadata that can be attached to an `ItemSchemeAssociation` and `ItemAssociation`. The `associationType` defines the role of the `ItemSchemeAssociation` and `ItemAssociation`. Note that the *`Item`* associated by the `associationType` is not in the same *`ItemScheme`* as the Items related by the `ItemAssociation` – it is in a specific scheme (code list) of role types.

866 **4.9.3.2 Definitions**

Class	**Feature**	**Description**
ItemSchemeAssociation	Inherits from *ItemScheme*	Associates two Item Schemes in a way defined by the association role.
	/source	Associates the source Item Scheme.
	/target	Associates the target Item Scheme.
	/items	Associates the Item Associations that each link to a source and a target Item.
	+associationType	This is a link to an Item in a “role” Item Scheme that defines the role of the Item Scheme Association.
	/properties	Associates Property to the Item Scheme Association
ItemAssociation	Inherits from *Item*	
	/source	Associates the source Item.
	/target	Associates the target Item.
	+associationType	This is a link to an Item in a “role” Item Scheme that defines the role of the Item Association.
	/properties	Associates Property to the Item Association

867

5 KEY FAMILY (DATA STRUCTURE DEFINITION) AND DATASET

5.1 Introduction

The `KeyFamily` is the class name for a structure definition for data. Many organisations know this type of definition a “Data Structure Definition” and so the two names are synonymous. The term Key Family is used in this specification.

Many of the constructs in this layer of the model inherit from the SDMX Base layer. Therefore, it is necessary to study both the inheritance and the relationship diagrams to understand the functionality of individual packages. In simple sub models these are shown in the same diagram, but are omitted from the more complex sub models for the sake of clarity. In these cases, the diagram below shows the full inheritance tree for the classes concerned with data structure definitions.

There are very few additional classes in this sub model other than those shown in the inheritance diagram below. In other words, the SDMX Base gives most of the structure of this sub model both in terms of associations and in terms of attributes. The relationship diagrams shown in this section show clearly when these associations are inherited from the SDMX Base (see the Appendix “A Short Guide to UML in the SDMX Information Model” to see the diagrammatic notation used to depict this).

The actual SDMX Base construct from which the concrete classes inherit depends upon the requirements of the class for:

- Annotation - *`AnnotableArtefact`*
- Identification - *`IdentifiableArtefact`*
- Versioning – *`VersionableArtefact`*
- Maintenance - *`MaintainableArtefact`*

5.2 Inheritance View

5.2.1 Class Diagram

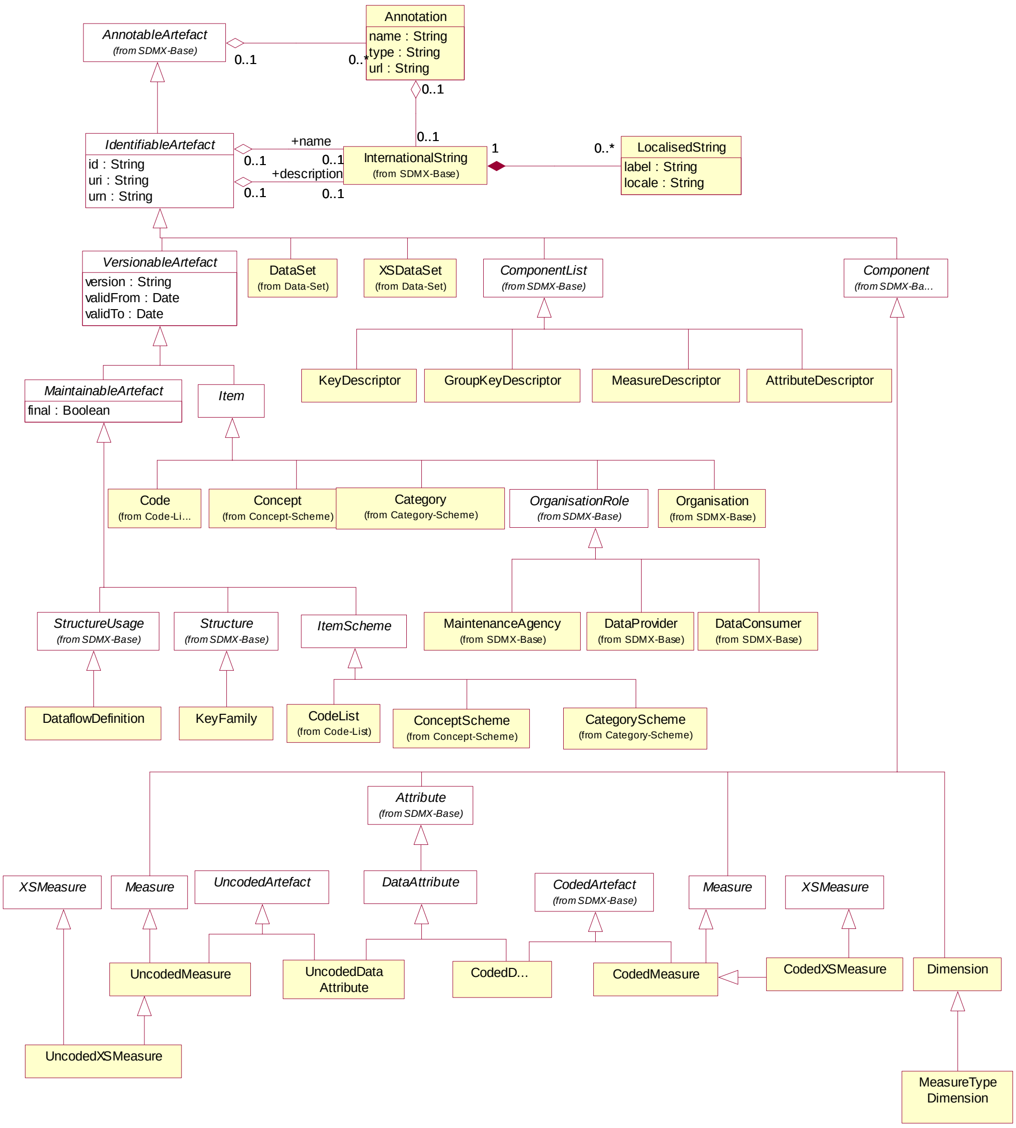


Figure 22 Class inheritance in the Key Family and Data Set packages

5.2.2 Explanation of the Diagram

5.2.2.1 Narrative

Those classes in the SDMX metamodel which require annotations inherit from *AnnotableArtefact*. These are:

- *IdentifiableArtefact*

Those classes in the SDMX metamodel which require annotations, global identity, multilingual name and multilingual description are derived from *IdentifiableArtefact* . These are:

- *VersionableArtefact*

The classes in the SDMX metamodel which requires annotations, global identity, multilingual name and multilingual description, and versioning are derived from *VersionableArtefact* . These are:

- *MaintainableArtefact*
- *Item*

Abstract classes which represent information that is maintained by Maintenance Agencies all inherit from *MaintainableArtefact*, they also inherit all the features of a *VersionableArtefact*, and are:

- *StructureUsage*
- *Structure*
- *ItemScheme*

All the above classes are abstract. What is of importance to understanding the class diagrams presented in this section are the concrete classes that inherit from these abstract classes.

Those concrete classes in the SDMX Key Family and Dataset packages of the metamodel which require to be maintained by Maintenance Agencies all inherit (via other abstract classes) from *MaintainableArtefact*, these are:

- DataflowDefinition
- KeyFamily

The component structures that are lists of lists, inherit directly from *Structure*. A *Structure* contains several lists of components (e.g. a KeyFamily contains a list of dimensions, a list of measures and a list of attributes). For key family (data structure) definitions the one concrete (structure) class for data structure definitions is:

- KeyFamily

The concrete classes which inherit from *ComponentList* and are sub components of the KeyFamily are:

- KeyDescriptor
- GroupKeyDescriptor

- `MeasureDescriptor`
- `AttributeDescriptor`

The classes that inherit from *Component* (i.e. these are the concrete components of the classes above) are:

- *Measure*
- `Dimension`
- *Attribute*

The Attribute has a further abstract class of:

- *DataAttribute*

The concrete classes which inherit from the abstract classes *Measure* and *DataAttribute* are:

- `CodedMeasure`
- `UncodedMeasure`
- `CodedDataAttribute`
- `UncodedDataAttribute`

Furthermore, the artefacts that are not coded (`UncodedDataAttribute` and `UncodedMeasure`) inherit from the *UncodedArtefact*, and those that are coded (`CodedDataAttribute` and `CodedMeasure`) inherit from *CodedArtefact*. The differences between a *CodedArtefact* and an *UncodedArtefact* (as detailed earlier in the explanation of the base structures) are:

- A *CodedArtefact* has an association to an *ItemScheme* which, in the context of the `KeyFamily` is its sub class `CodeList`
- The *UncodedArtefact* has no such association but has additional attributes to describe its format and type

Cross sectional measures are sub classes of the time series measures and of a common abstract class *XSMeasure*

- `UncodedXSMeasure` inherits from `UncodedMeasure` and *XSMeasure*
- `CodedMeasure` inherits from `CodedMeasure` and *XSMeasure*

Finally, the `MeasureTypeDimension` is sub class of `Dimension` as it has specific associations in addition to those for the `Dimension` itself (see the relationship diagram below). With the exception of `MeasureTypeDimension` the specific roles played by Dimensions are supported by an association to a role and are not depicted as sub classes.

The concrete classes identified above are all of the classes required to define the metamodel for the KeyFamily. The diagrams and explanations in the rest of this section show how these concrete classes are related so as to support the functionality required.

5.3 Key Family – Relationship View

5.3.1 Class Diagram

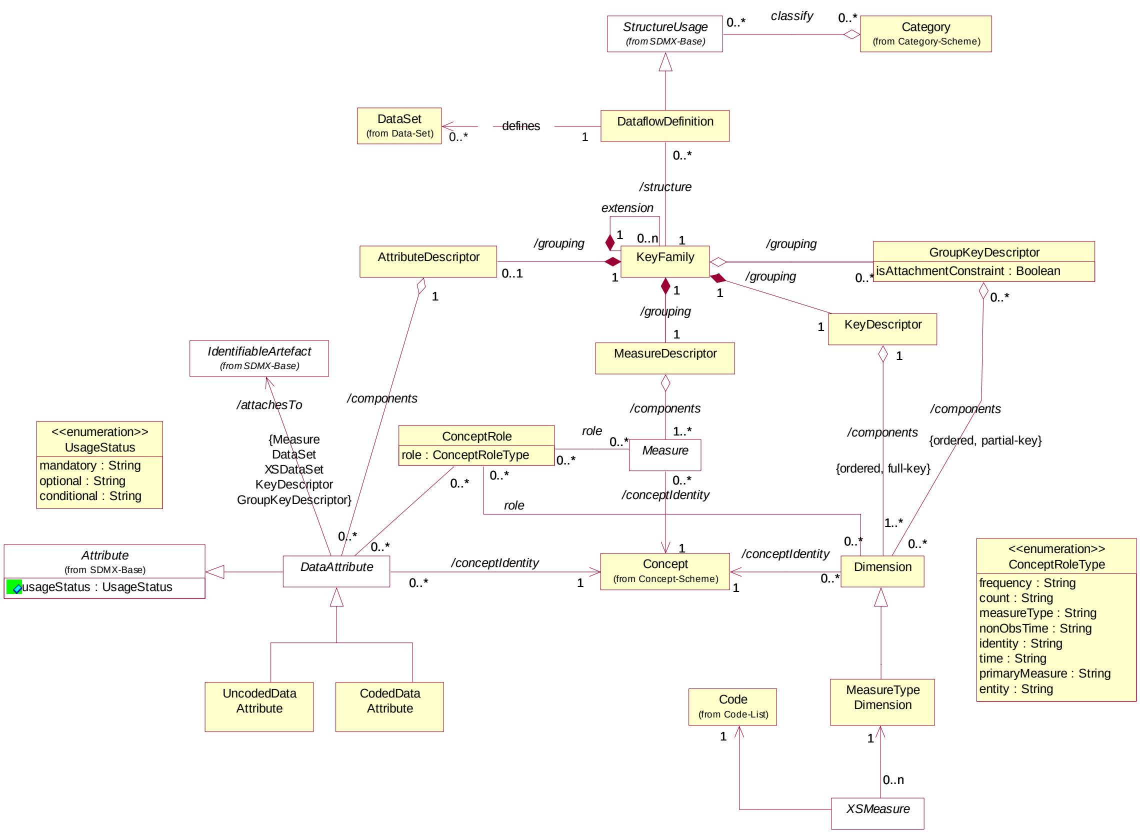


Figure 23 Relationship class diagram of the Key Family excluding representation

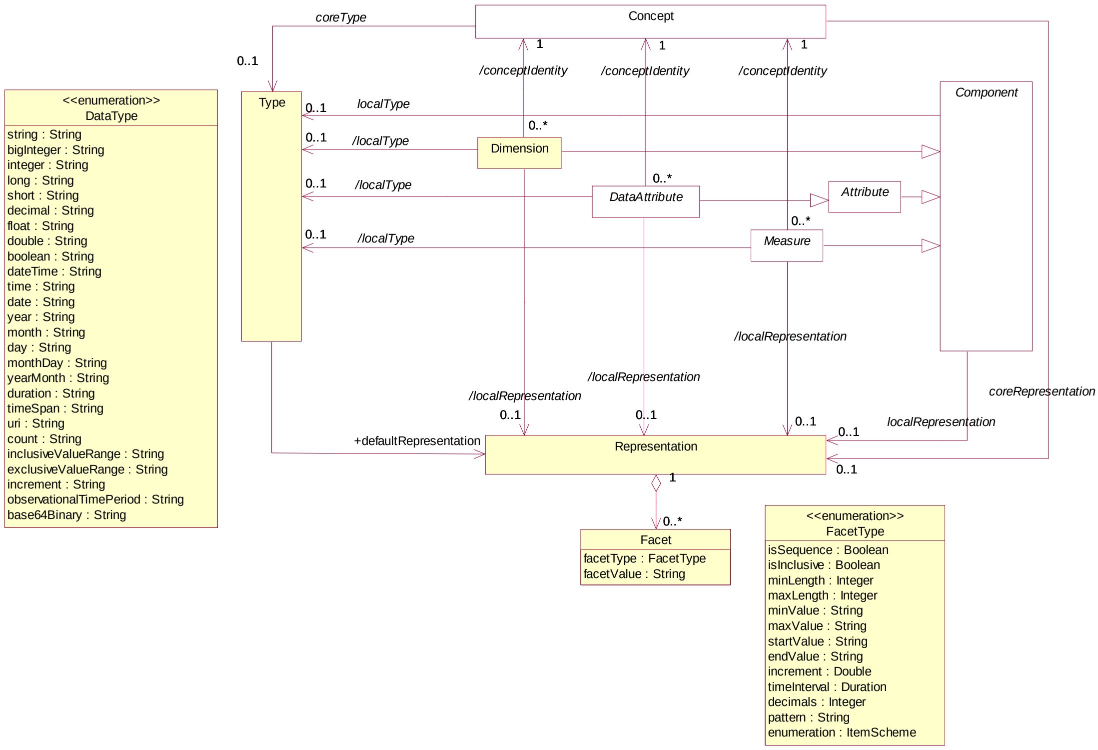


Figure 24 Relationship class diagram of the Key Family representation

5.3.2 Explanation of the Diagrams

5.3.2.1 Narrative

A `KeyFamily` defines the `Dimensions`, *`DataAttribute`*s, *`Measure`*s, and associated `Representation` that comprise the valid structure of data and related metadata that are contained in a `DataSet`, which is defined by a `DataflowDefinition`.

The `DataflowDefinition` associates a `KeyFamily` with one or more `Category` (possibly from different `CategorySchemes`) via the parent class of `DataflowDefinition` - *`StructureUsage`*. This gives a system the ability to state which `DataSets` are to be reported/disseminated for a given `Category`, and which `DataSets` can be reported using the `KeyFamily` definition. The `DataflowDefinition` may also have additional metadata attached that defines qualitative information and constraints on the use of the `KeyFamily` such as the sub set of `Codes` used in a `Dimension` (this is covered later in this document – see "Data Constraints and Provisioning" section 9). Each `DataflowDefinition` must have one `KeyFamily` specified which defines the structure of any `DataSets` to be reported/disseminated.

`Dimension`, *`DataAttribute`*, and *`Measure`* each link to the `Concept` that defines its name and semantic. The valid values for a `Dimension`, `Measure`, or *`DataAttribute`*, when used in this `KeyFamily`, are defined by the `Representation`. This `Representation` is taken from the `Concept` definition (coreRepresentation) unless it is overridden in this KeyFamily (localRepresentation).

The Dimension can be grouped in two ways:

1. There will always be a KeyDescriptor grouping that identifies all of the Dimensions comprising the full key.

2. Optionally there may be multiple GroupKeyDescriptors each of which identifies the group of Dimensions that can form a partial key. The GroupKeyDescriptor must be identified (GroupKeyDescriptor.id) and is used in the GroupKey of the DataSet to group sets of full keys to which a *DataAttribute* can be attached.

The *Measure* is the observable phenomenon and the set of *Measure*s in the KeyFamily is grouped by a single MeasureDescriptor. A *Measure* can be coded (CodedMeasure) or un-coded (UncodedMeasure) - these concrete sub classes of Measure are not shown on the diagram.

The *DataAttribute* defines a characteristic of data that are collected or disseminated and is grouped in the KeyFamily by a single AttributeDescriptor. The *DataAttribute* can be specified as being mandatory, conditional, or optional (as defined in usageStatus – inherited from the parent *Attribute* class).

The *DataAttribute* is an abstract class and is either a CodedDataAttribute or an UncodedDataAttribute.

A *DataAttribute* is specified as being "attachable to" a part of the structure of the KeyFamily. The *DataAttribute* can be specified as being attachable to a constrained set of *IdentifiableArtefact*s. The constrained set is as follows:

- Measure
- DataSet
- XSDataSet
- KeyDescriptor
- GroupKeyDescriptor

It is possible to specify that a *DataAttribute* is attached to a sub set of the series keys or sub set of the possible values that a component can take (such as a Dimension). This is specified by declaring in the GroupKeyDescriptor that there is an AttachmentConstraint (isAttachmentConstraint) that specifies this sub set. The Id of the AttachmentConstraint is the same as the Id of the GroupKeyDescriptor. AttachmentContraints are described in section 10.3. If there is an AttachmentConstraint then the GroupKeyDescriptor does not specify any Dimensions, as the dimensionality of the constraint is defined in the AttachmentConstraint.

The valid structures for a KeyFamily definition to which a *DataAttribute* can be specified as being attachable, and actual structure in the DataSet to which the AttributeValue is attached are:

- DataSet and XSDataSet – AttributeValue attached to DataSet or XSDataSet
- GroupKeyDescriptor (identified in addition by the GroupKeyDescriptor.id) - AttributeValue attached to GroupKey, Group, Section
- KeyDescriptor – AttributeValue attached to TimeSeriesKey
- *Measure* - AttributeValue attached to Observation or XSObservation

If there is a requirement to attach metadata to other KeyFamily artefacts such as Dimension, or even the KeyFamily itself, or to slices of the data cube for which no AttachmentConstraint was specified in the KeyFamily itself, then these can be specified in the Metadata Structure Definition, which is explained later.

The Concepts used for each of Dimension, *Measure*, and *DataAttribute* can play a specific role in the KeyFamily, and the association to the ConceptRole supports this. The roles are constrained to those in the datatype ConceptRoleType and each component type is constrained by the roles it can play as shown in the table below.

Role	**Description**	**Valid for component type**	**Role be played by multiple components**
frequency	identifies the Concept that plays the role of frequency	Dimension DataAttribute	No
count	identifies the Concept that plays the role of an identifier where the identifier is taken from a known system of counts	Dimension DataAttribute	Yes
measureType	identifies the Concept that plays the role of identifying a type of measure	Dimension	Yes
entity	identifies the Concept that plays the role of the subject to whom the data refers (e.g. the reporting agent for primary	Dimension DataAttribute	No

Role	Description	Valid for component type	Role be played by multiple components
	reporting, the country for secondary reporting)		
time	identifies the `Concept` that specifies the time of the observation of the `primaryMeasure`	Dimension	No
nonObsTime	identifies the `Concept` that plays the role of a date/time identifier in the `KeyFamily` which is not related to the time of the observation	Dimension DataAttribute	Yes
primaryMeasure	identifies the `Concept` that plays the role of the observation in a time series	Measure	No
identity	identifies the `Concept` that plays the role of an identifier which is taken from a known scheme of identifiers.	Dimension DataAttribute	Yes

Each of `Dimension`, `Measure`, and *`DataAttribute`* can have a `Type` and *`Representation`* specified (using the `localType` and `localRepresentation` associations). If this is not specified in the `KeyFamily` definition then the `Type` and *`Representation`* is taken from that defined for the `Concept` (the `coreType` and `coreRepresentation` associations). Whilst the class diagram in Figure 24 looks complex it is effectively portraying:

1. The `Concept` has an association to *`Representation`* (`coreRepresentation`) and to `Type` (`coreType`)
2. The *`Component`* has an association to *`Representation`* (`localRepresentation`) and to `Type` (`localType`).
3. The `Dimension`, `DataAttribute`, and `Measure` all inherit from *`Component`* and therefore inherit the `localRepresentation` and `localType` associations – shown on the diagram as an inherited associations (`/localRepresentation, /localType`)

The definition of the various types of *`Facet`* and the `Type` can be found in section 4.4.

The `MeasureTypeDimension` associates the `CodeList` whose `Codes` will become the *`XSMeasure`*s in a cross sectional key family, and supports the transformation of a cross sectional data set to a time series data set and also vice versa: the `Concepts` that are the *`XSMeasure`*s in a cross sectional key family are the `Codes` in the `CodeList` associated to the `MeasureTypeDimension`. Each *`XSMeasure`* has a uni-directional association to a MeasureTypeDimension and to a Code. This Code is contained in the CodeList associated to the MeasureTypeDimension. There can be more than one MeasureTypeDimension in a KeyFamily.

Furthermore, the CodeList attached to each of CodedDataAttribute that define the measurement characteristics (such as unit of measure) of each of the *XSMeasure*s in a cross sectional data set are concatenated into a single CodeList that define the measurement characteristics of the relevant *Measure* in the equivalent time series.

For example, if there are three *XSMeasure* Concepts called *Weight*, *Value*, and *Volume* then when transformed into a time series the *XSMeasure* Concepts become an additional Dimension (MeasureTypeDimension) with three values in the associated CodeList (*weight, value, volume*). The (now) single Measure in the time series may have a *Unit_Of_Measure* CodedAttribute which is associated to a CodeList: this CodeList must have all of the values of the three CodeList used for the three *XSMeasure*s.

A KeyFamily definition can be extended to form a derived KeyFamily. The extension of a KeyFamily is limited to:

- The addition of Dimensions, *DataAttribute*s, and *Measure*s
- The specification of additional of GroupDescriptors
- The change of usageStatus for a *DataAttribute*
- The change of CodeList used for a Dimension or *DataAttribute*
- The change of a *DataAttribute* from CodedDataAttribute to UncodedDataAttribute or vice-versa

5.3.2.2 Definitions

Class	**Feature**	**Description**
StructureUsage		See “SDMX Base”.
	classify	Associates one or more Categories in one or more schemes that define data categorisation in terms of data to be reported or data to be disseminated.
Category		See “Category Scheme”.
DataflowDefinition	Inherits from *StructureUsage*	Abstract concept (i.e. the structure without any data) of a flow of data that providers will provide for different

Class	Feature	Description
		reference periods.
	structure	Associates a data flow definition to the Key Family.
KeyFamily		A collection of metadata concepts, their structure and usage when used to collect or disseminate data.
	/grouping	An association to a set of metadata concepts that have an identified structural role in a Key Family.
	classify	Associates the Category by which this Dataflow is classified.
GroupKeyDescriptor	Inherits from *ComponentList*	A set metadata concepts that define a partial key derived from the Key Descriptor in a Key Family.
	isAttachment Constraint	Specifies whether there is an Attachment Constraint that specifies the sub set of Dimension, Measure, or Attribute values to which an Attribute can be attached.
	/components	An association to a component in a set of components.
KeyDescriptor	Inherits from *ComponentList*	An ordered set of metadata concepts that, combined, classify a statistical series, such as a time series, and whose values, when combined (the key) in an instance such as a data set, uniquely identify a specific series.
	/components	An association to a component in a set of components.
AttributeDescriptor	Inherits from	A set metadata concepts that define the attributes

Class	Feature	Description
	ComponentList	of a key family.
	/components	An association to a component in a set of components.
MeasureDescriptor	Inherits from *ComponentList*	A set metadata concepts that define the measures of a key family.
	/components	An association to a component in a set of components.
Dimension	Inherits from *Component* Sub classes *MeasureTypeDimension*	A statistical concept used (most probably together with other statistical concepts) to identify a statistical series, such as a time series, e.g. a statistical concept indicating a certain economic activity or a geographical reference area.
	/conceptIdentity	An association to the metadata concept which defines the semantic of the component.
	/localType	Associates a Type (data type) that overrides any core type specified for the Concept itself.
	/localRepresentation	Associates a Representation that overrides any core representation specified for the Concept itself.
MeasureTypeDimension	Inherits from Dimension	A metadata concept used to refer to and identify a dimension in a time series that defines the concepts for the Measure when cross sectional data is represented in a time series.
DataAttribute	Abstract class Sub classes: *CodedDataAttribute* *UncodedDataAttribute*	A characteristic of an object or entity.
	/localType	Associates a Type (data

Class	Feature	Description
		type) that overrides any core type specified for the Concept itself.
	/localRepresentation	Associates a Representation that overrides any core representation specified for the Concept itself.
UncodedDataAttribute	Inherits from *DataAttribute* *CodedArtefact*	A characteristic of an object or entity that has a free text representation.
CodedDataAttribute	Inherits from *DataAttribute* *UncodedArtefact*	A characteristic of an object or entity that takes its values from a code list.
Measure	Inherits from *Component* Sub classes: *CodedMeasure* *UncodedMeasure*	The concept that is the phenomenon to be measured in a time series data set. In a data set the instance of the measure is often called the observation.
	/localType	Associates a Type (data type) that overrides any core type specified for the Concept itself.
	/localRepresentation	Associates a Representation that overrides any core representation specified for the Concept itself.
CodedMeasure	Inherits from *Measure* Sub classes: *CodedXSMeasure*	A time series Measure that is coded.
UncodedMeasure	Inherits from *Measure* Sub classes: *UncodedXSMeasure*	A time series Measure that is un-coded.
CodedXSMeasure	Inherits from CodedMeasure *XSMeasure*	A cross sectional Measure that is coded.
UncodedMeasure	Inherits from Measure	A cross sectional Measure that is un-coded.

Class	Feature	Description
	XSMeasure	
XSMeasure		The phenomenon to be measured in a cross sectional data set.
ConceptRole		Specifies the role that a concept plays when it is used in a component of a structure, such as a Dimension in a Key Family.
	role	Identifies the specific role.

5.4 Data Set – Timeseries Relationship View

5.4.1 Context

A data set comprises the collection of data values and associated metadata that are collected or disseminated according to a known key family definition.

5.4.2 Class Diagram

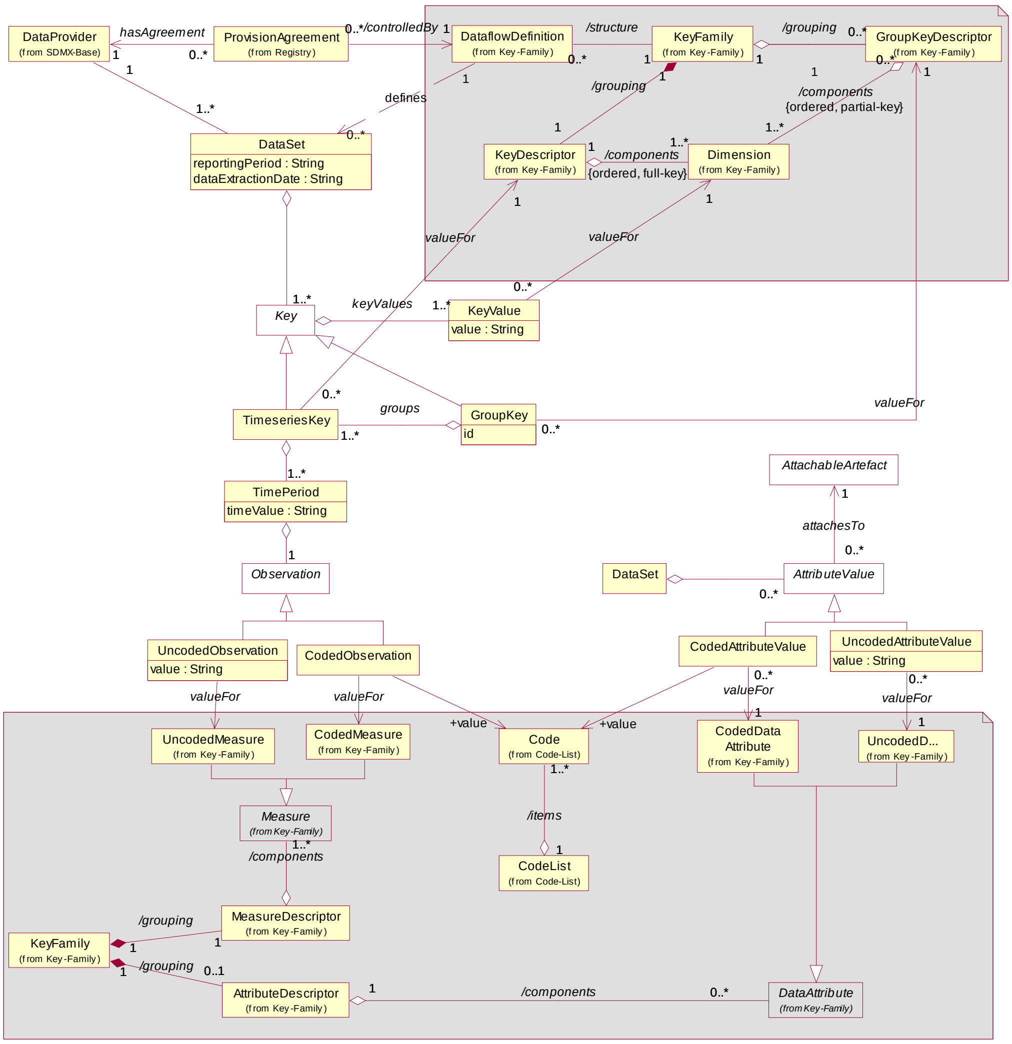


Figure 25 Class diagram of the time series Data Set

5.4.3 Explanation of the Diagram

5.4.3.1 Narrative

Note that the `DataSet` must conform to the `KeyFamily` definition associated to the `DataflowDefinition` for which this `DataSet` is an "instance of data". Whilst the model shows the association to the classes of the `KeyFamily`, this is for conceptual purposes to show the link to the `KeyFamily`. In the actual `DataSet` as exchanged there must, of course, be a reference to the `DataflowDefinition`, but the `KeyFamily` definition is not necessarily exchanged with the data. Therefore, the `KeyFamily` classes are shown in the grey areas, as these are not a part of the `DataSet` itself.

An organisation in the role of DataProvider can be responsible for one or more DataSet. The DataProvider may have a DataflowAgreement that links to the DataflowDefinition for which this DataSet is being provided. DataflowAgreement and DataflowDefinition are described later in the section on Data Provision.

A timeseries DataSet is a collection of a set of Observations that share the same dimensionality, which is specified by a set of unique Dimension defined in the KeyDescriptor of the KeyFamily, together with associated *AttributeValue*s that define specific characteristics about the Observation, Key, or DataSet.

For timeseries each unique combination of KeyValue (TimeseriesKey) combined with a TimePeriod, identifies precisely one Observation.

The *Observation* is the value of the variable being measured for the Concept associated to the *Measure* in the MeasureDescriptor of the KeyFamily. The *Observation* can relate to CodedMeasure – this is the CodedObservation – or to an UncodedMeasure – this is the UncodedObservation.

The GroupKey is a sub unit of the Key that has the same dimensionality as the TimeseriesKey, but defines a subset of the KeyValues of the TimeseriesKey. Its sub dimension structure is defined in the GroupKeyDescriptor of the KeyFamily identified by the same id as the GroupKey. The id identifies a “type” of group and the purpose of the GroupKey is to identify a set of individual TimeseriesKey so that one or more AttributeValue can be attached at this group level. There can be many types of groups in a DataSet.

Each of DataSet, TimeseriesKey, GroupKey, and Observation can have zero or more AttributeValue that defines some metadata about the object to which it is associated. The allowable Concepts and the objects to which these metadata can be associated (attached) are defined in the KeyFamily. The link to the object in the DataSet is shown by the association to AttachableArtefact. The diagram below shows the object types to which the AttributeValue can be attached.

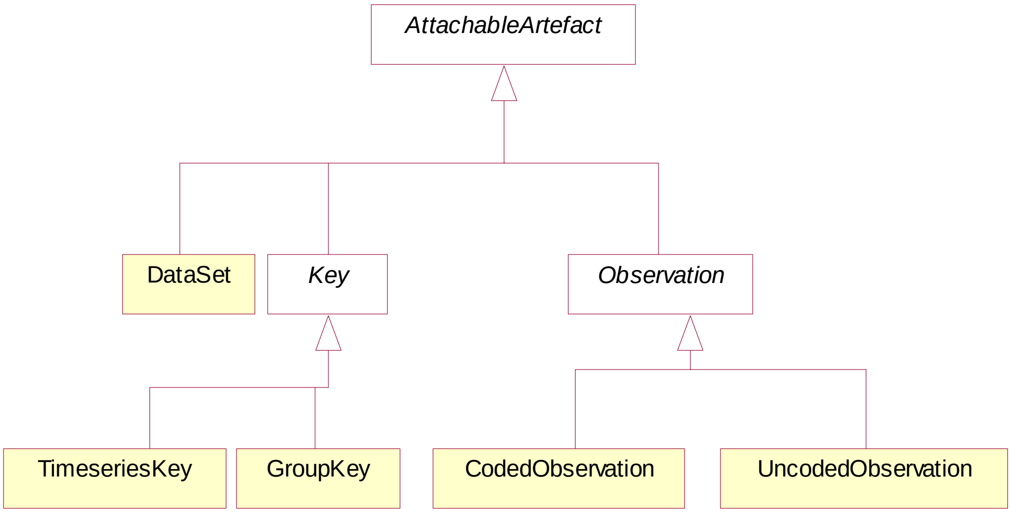


Figure 26: Attribute Value attachment for a time series data set

The `AttributeValue` therefore links to the object type (`DataSet`, `TimeseriesKey`, `GroupKey`, `CodedObservation`, `UncodedObservation`) and the actual object as identified by its key (e.g. the `DataSet`, `KeyValues` of the `TimeseriesKey` or `GroupKey`, or `Observation` (`TimeseriesKey` plus `TimePeriod`).

5.4.3.2 Definitions

Class	Feature	Description
`DataSet`		An organised collection of data.
	`reportingPeriod`	A specific time period in a known system of time periods that identifies the period of a report.
	`dataExtractionDate`	A specific time period that identifies the date and time that the data are extracted from a data source.
	`describedBy`	Associates a data flow definition and thereby a Key Family to the data set.

`Key`	Abstract class
	Sub classes
	`TimeseriesKey`
	`GroupKey`
		Comprises the cross product of values of dimensions that identify uniquely a statistical series such as a time series.

	`keyValues`	Associates the individual Key Values that comprise the Key.

Class	Feature	Description
`KeyValue`		The value of a component of a key such as the value of the instance a Dimension in a multidimensional structure, like the Key Descriptor of a Key Family.
	`value`	The value of the key component.
	`valueFor`	Associates a dimension to the Key Value, and thereby to the Concept that is the semantic of the Dimension.
`GroupKey`	Inherits from `Key`	A set of Key Values that comprise a partial key, of the same dimensionality as the Time Series Key, and which group together a set of series keys (i.e. the scope of the Timeseries Keys identified by the Group Key is defined using the same Dimensions as the Timeseries Key).
	`valueFor`	Associates the group key descriptor defined in the key family.
	`groups`	Associates a set of Time Series Keys.
`TimeseriesKey`	Inherits from `Key`	Comprises the cross product of values of all the dimensions that identify uniquely a time series.
`TimePeriod`		A specific time period in a known system of time periods.
	`timeValue`	The value of a time period.

`Observation`	Abstract class
	
	Sub classes
	`UncodedObservation`
	`CodedObservation`
		The value, at a particular period, of a particular variable.

`UncodedObservation`	Inherits from *`Observation`*	An observation that has a text value.

Class	Feature	Description
	value	The text value of the observation.
	valueFor	Associates the uncoded measure defined in the Key Family.
CodedObservation	Inherits from *Observation*	An Observation that takes it value from a code in Code List.
	valueFor	Associates the Coded Measure defined in the Key Family.
	+value	Association to the Code that is the value of the Observation.
AttributeValue	Abstract class Sub classes *UncodedAttributeValue* *CodedAttributeValue*	The value of an attribute, such as the instance of a Coded Attribute or of an Uncoded Attribute in a structure such as a Key Family.
	attachesTo	Associates the attribute to the object to which it is attached.
AttachableArtefact		The object to which the attribute value is attached.
UncodedAttributeValue	Inherits from *AttributeValue*	An attribute value that has a text value.
	value	The text value of the attribute.
	valueFor	Associates the Coded Data Attribute defined in the Key Family.
CodedAttributeValue	Inherits from *AttributeValue*	An attribute that takes it value from a Code in Code List.
	valueFor	Associates the Uncoded Data Attribute defined in the Key Family.
	+value	Association to the Code that is the value of the Observation.

1197

5.5 Data Set – Cross Sectional Relationship View

5.5.1 Class Diagram

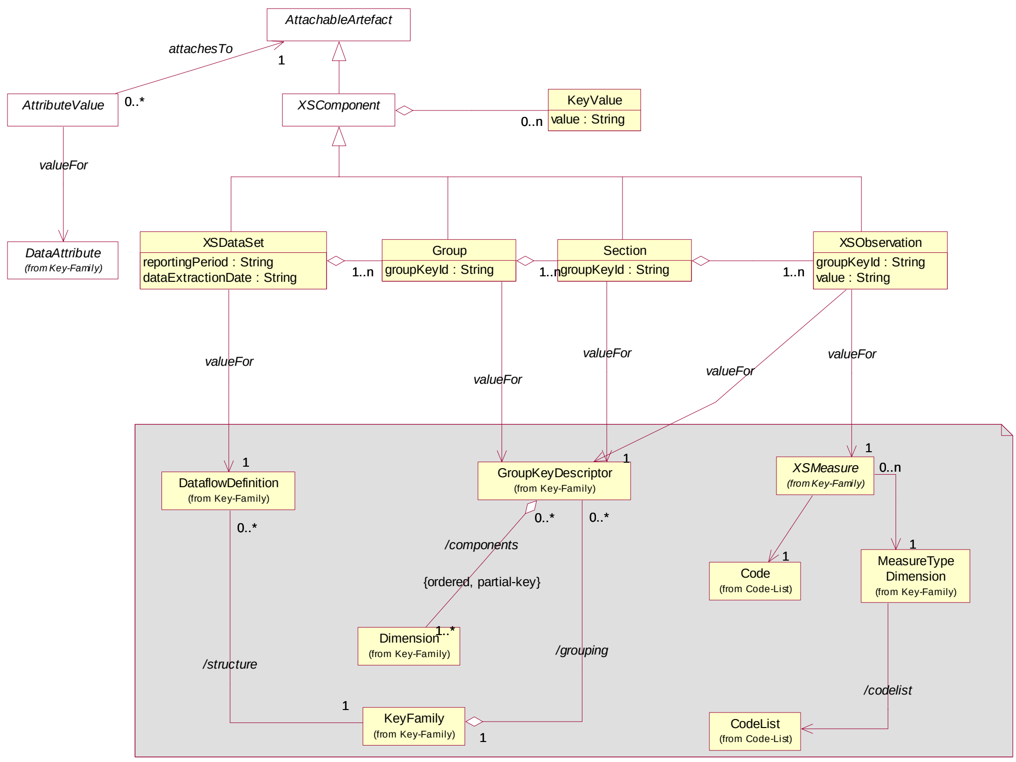


Figure 27 Class diagram of the cross sectional Data Set

5.5.2 Explanation of the Diagram

5.5.2.1 Narrative

The cross sectional data set – `XSDataSet` - differs from the timeseries `DataSet` in the following ways:

1. There is no "full key" specified and so there is no concept of a "cross sectional key" as there is the concept of a timeseries key in the time series data set: cross sectional data are by their nature identified by one or more partial keys which together comprise the "full key".

2. The meaning of "group" is therefore different from the timeseries: in a timeseries the `GroupKey` serves to group individual timeseries so that common attributes can be attached. The role of the `Group` in the cross sectional data set is twofold: it describes a partial key (which must be combined with the keys in the subordinate components in order to fully identify the observation); and it is a structure to which attributes can be attached.

3. The Dimension values (KeyValue) can be expressed in one of the three levels in the structure: GroupKey, Section, and XSObservation. Therefore, partial keys can be declared at each of these levels which, together, make up the full key.

4. Similarly, AttributeValues can be associated at any of the three levels, plus the level of the XSDataSet itself.

5. If time is present in the XSDataSet then it is expressed at the level of the Group.

Note that the KeyFamily definition does not need to prescribe that a particular Dimension or Attribute is reported at a particular level: indeed it is the nature of many cross sectional series to leave this aspect dynamic. The minimal pre-requisites in the KeyFamily definition to support the cross sectional data set are:

- to declare a GroupKeyDescriptor that contains all of the Dimensions
- to make all of the MetadataAttributes attachable at this group level.

Clearly, the KeyFamily definition can be more prescriptive and define the precise contents of for each of Group, Section, and XSObservation by declaring many GroupKeyDescriptors, each one individually identified by the GroupKeyDescriptor.id.

The identity of the XSObservation is taken from a Code in the CodeList used by the MeasureTypeDimension in the KeyFamily definition. There can be many XSObservation in a Section, each one containing the reported value for one of the Codes (note that each can also identify KeyValues and AttributeValues as mentioned above).

The association to the KeyFamily constructs is shown by the classes in the grey box. As with the timeseries DataSet, there will be a reference to the DataFlowDefinition in the XSDataSet.

5.5.2.2 Definitions

Class	Feature	Description
XSComponent	Abstract class Sub classes are: DataSet Group Section XSObservation	

Class	**Feature**	**Description**
`KeyValue`		The value of a component of a key such as the value of the instance a Dimension in a multidimensional structure, like the Key Descriptor of a Key Family.
`XSDataSet`		An organised collection of cross sectional data
`Group`	Inherits from `XSComponent`	A set of key values that comprise a partial key, of the same dimensionality as the full key, and which group together a set of sections (ie, the scope of the Section grouped by the Group is defined using a partial set of the same Dimensions as defined in the full key).
	valueFor	Associates the GroupKeyDescriptor that defines the partial key.
Section	Inherits from `XSComponent`	A set of key values that comprise a partial key, of the same dimensionality as the full key, and which group together a set of cross sectional obervations (ie, the scope of the XSObservation grouped by the Section is defined using a partial set of the same Dimensions as defined in the full key).
	`valueFor`	Associates the `GroupKeyDescriptor` that defines the partial key.
`XSObservation`	Inherits from `XSComponent`	An observation in a cross sectional data set that optionally defines a set of key values that comprise a partial key, of the same dimensionality as the full key.

Class	Feature	Description
	valueFor (XSMeasure)	Associates the XSMeasure that is the concept of the observation.
	valueFor (GroupKeyDescriptor)	Associates the GroupKeyDescriptor that defines the partial key

1252

6 CUBE

6.1 Context

Some statistical systems create views of data based on a "cube" structure. In essence, a cube is an n-dimensional object where the value of each dimension can be derived from a hierarchical code list. The utility of such cube systems is that it is possible to "roll up" or "drill down" each of the hierarchy levels for each of the dimensions to specify the level of granularity required to give a "view" of the data – some dimensions may be rolled up, others may be drilled down. Such systems give a dynamic view of the data, with aggregated values for rolled up dimension positions. For example, the individual countries may be rolled up into an economic region such as the EU, or a geographical region such as Europe, whilst another dimension, such as "type of road" may be drilled down to its lower level. The resulting measure (such as "number of accidents") would then be an aggregation of the value for each individual country for the specific type of road.

Such cube systems rely, not on simple code lists, but on hierarchical code sets (see section 8).

6.2 Support for the Cube in the Information Model

Data reported using a key family structure (where each dimension value, if coded, is taken from a flat code list) can be described by a cube definition and can be processed by cube aware systems. The SDMX-IM supports the definition of such cubes in the following way:

- The `HierachicalCodeScheme` defines the (often complex) hierarchies of codes
- The `StructureSet`
    - groups `KeyFamily` that describe the cube
    - provides a mapping mechanism between the codes in the flat code lists used by the `KeyFamily` and a `HierarchicalCodeScheme`

7 METADATA STRUCTURE DEFINITION AND METADATA SET

7.1 Context

The SDMX metamodel allows metadata:

1. To be exchanged without the need to embed it within the object that it is describing.

2. To be stored separately from the object that it describes, yet be linked to it (for example, an organisation has a metadata repository which supports the dissemination of metadata resulting from metadata requests generated by systems or services that have access to the object for which the metadata pertains).

3. To be indexed to aid searching (example: a registry service can process a metadata report and extract structural information that allows it to catalogue the metadata in a way that will enable users to query for it).

4. To be reported according to a defined structure.

In order to achieve this, the following structures are modelled

- metadata structure definition which has the following components:
    - the object types to which the metadata are to be associated (attached)
    - the components that, together, comprise a unique key of the object type
    - the reporting structure comprising the metadata attributes that can be attached to the various object types (these attributes can be structure din a hierarchy), together with any constraints that may apply (e.g. association to a code list that contains valid values for the attribute when reported in a metadata set)
- the metadata set, which contains reported metadata

7.2 Inheritance

7.2.1 Introduction

As with the Structure Definitions, many of the constructs in this layer of the model inherit from the SDMX Base layer. Therefore, it is necessary to study both the inheritance and the relationship diagrams to understand the functionality of individual packages. The diagram below shows the full inheritance tree for the classes concerned with the `MetadataStructureDefinition` and the `MetadataSet`. The diagram does not include the classes already described but which are used in the reference metadata models (see 8.3.2).

There are very few additional classes in the `MetadataStructureDefinition` package that do not themselves inherit from classes in the SDMX base. In other words, the SDMX Base gives most of the structure of this sub model both in terms of associations and in terms of attributes. The relationship diagrams shown in this section show clearly when these associations are inherited from the SDMX Base (see the Appendix “A Short Guide to UML in the SDMX Information Model” to see the diagrammatic notation used to depict this). It is important to note that SDMX base structures used for the `MetadataStructureDefinition` are the same as those used for the `KeyFamily` and so, even though the usage is slightly different, the underlying way of defining a `MetadataStructureDefinition` is similar to that used for defining a `KeyFamily`.

The actual SDMX Base construct from which the concrete classes inherit depends upon the requirements of the class for:

- Annotation - *AnnotableArtefact*
- Identification - *IdentifiableArtefact*
- Versioning – *VersionableArtefact*
- Maintenance - *MaintainableArtefact*
- Ability to have additional dynamically defined metadata attached - *AttachableArtefact*

7.2.2 Inheritance Class Diagram

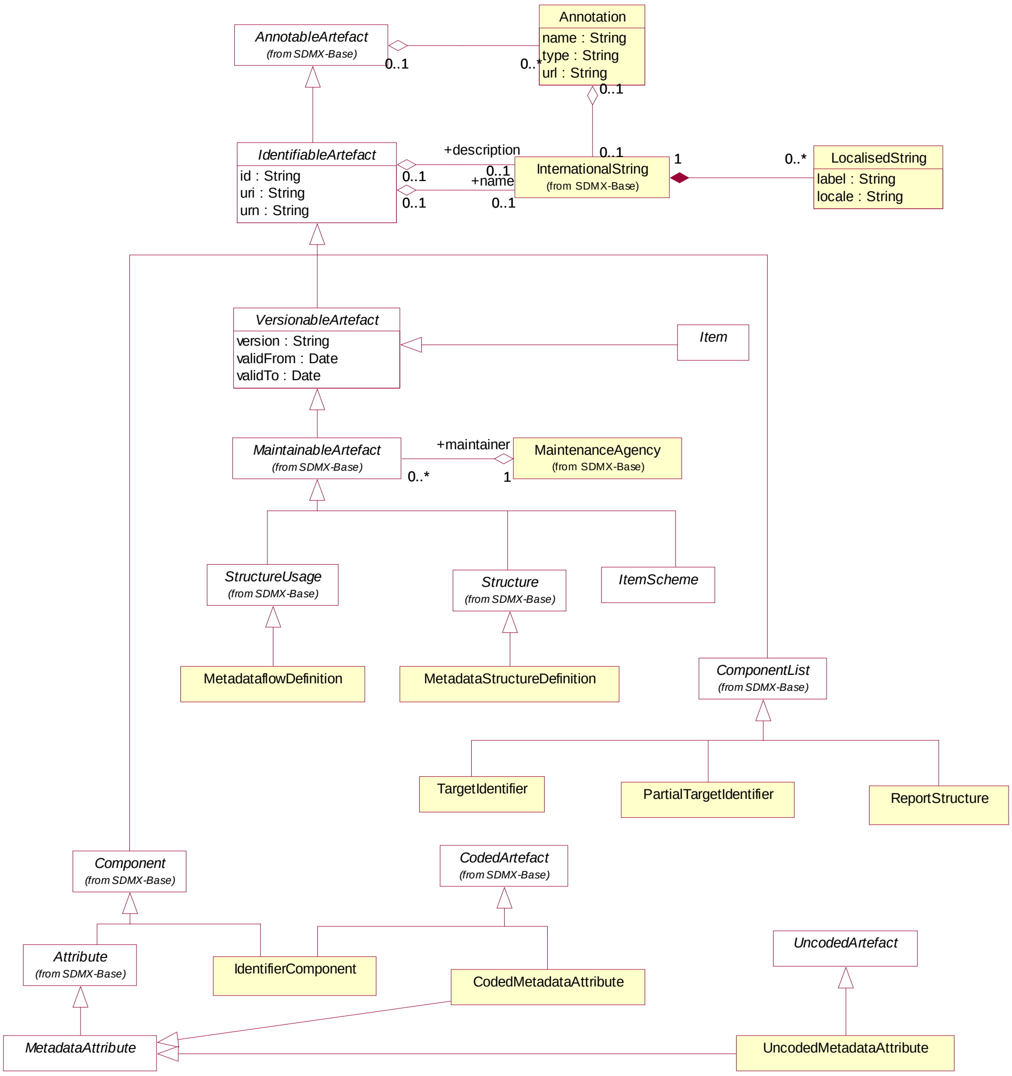


Figure 28: Class inheritance in the Metadata Structure Definition and Metadata Set packages

7.2.3 Explanation of the Diagram

7.2.3.1 Narrative

It is important to the understanding of the relationship class diagrams presented in this section to identify the concrete classes that inherit from the abstract classes.

The concrete classes in this part of the SDMX metamodel which require to be maintained by Maintenance Agencies all inherit from `MaintainableArtefact`, these are:

- *`StructureUsage`* (concrete class is `MetadataflowDefinition`)
- *`Structure`* (concrete class is `MetadataStructureDefinition`)

These classes also inherit the identity and versioning facets of *IdentifiableArtefact* and *VersionableArtefact*.

A *Structure* contains several lists of components. The concrete classes which inherit from *ComponentList* and in themselves are sub components of the `MetadataStructureDefinition` are:

- `TargetIdentifier`
- `PartialTargetIdentifier`
- `ReportStructure`

ComponentList contains `Components`. The classes that inherit from *Component* are:

- `IdentifierComponent`
- *MetadataAttribute*

The class which inherits from the abstract class *Attribute* that is relevant to the reference metadata and metadata set models is:

- *MetadataAttribute*

The *MetadataAttribute* is an abstract class and has two concrete sub classes:

- `CodedMetadataAttribute`
- `UncodedMetadataAttribute`

In addition to the inheritance from *MetadataAttribute* the `CodedMetadataAttribute` inherits from *CodedArtefact* and the `UncodedMetadataAttribute` inherits from *UncodedArtefact*.

7.3 Metadata Structure Definition

7.3.1 Introduction

With just one exception, the concrete classes identified above are all of the classes required to define the metamodel for metadata structure definitions. The diagrams and explanations in the rest of this section show how these concrete classes are related so as to support the functionality required. The exception is the `AttributeProperty` which does not inherit from any of the SDMX Base classes.

7.3.2 Structures Already Described

The `MetadataStructureDefinition` makes use of the following *ItemScheme* structures either as explicit concrete classes in the model, or as possible lists which comprise the value domain of an `IdentifierComponent`.

- CategoryScheme
- ConceptScheme
- CodeList
- OrganisationScheme

7.3.3 Class Diagram

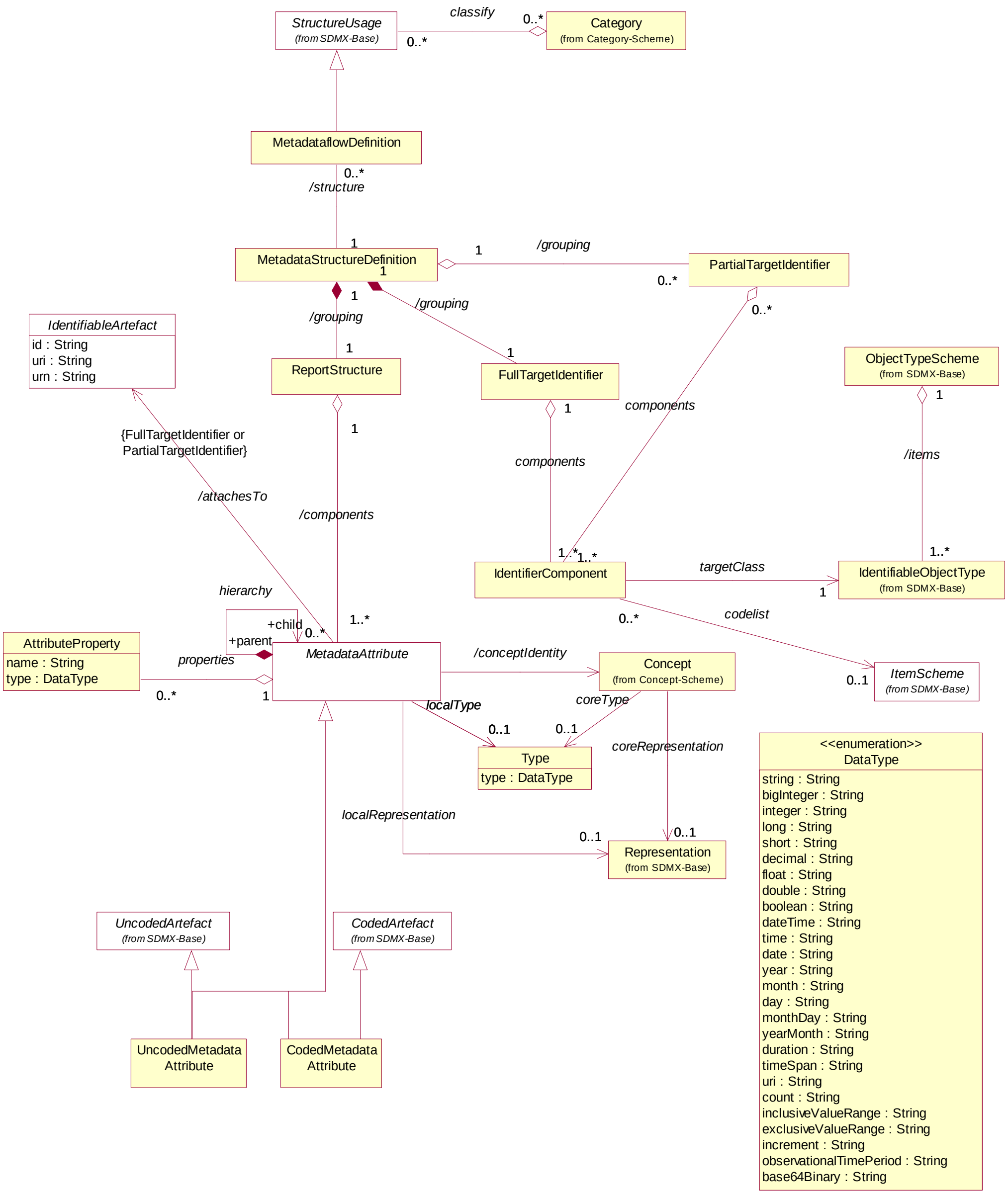


Figure 29: Relationship class diagram of the Metadata Structure Definition

7.3.4 Explanation of the Diagram

7.3.4.1 Narrative

In brief a `MetadataStructureDefinition` defines:

- The object type to which metadata can be associated (`IdentfiableArtefactType`).
- The components (`IdentifierComponent`) comprising the object identifier of the target object (`FullTargetidentifier` and `PartialTargetIdentifier`).
- The `ReportStructure` comprising the *`MetadataAttribute`*s that can be associated with the object type, and hierarchical structure of the attributes

The `FullTargetIdentifier` comprises on or more `IdentifierComponents` which, together comprise the scope of the `MetadataStructureDefinition` in terms of the object types that can be identified using this definition. Each `IdentifierComponent` must be associated to a `IdentifiableArtefactType` which itself may be taken from maintained scheme of `ObjectTypes`. In the context of this information model the `ObjectTypes` will be any class or group of classes (as defined by the `IdentifierComponents`) in the model that have identity, as it is instances of these object types or groups of object types to which metadata can be attached in a `MetadataSet`.

Instances of `IdentifierComponents` (i.e. the actual `IdentifierComponentValue` defined in a `MetadataSet`) are maintained in an *`ItemScheme`* (or, more precisely, a concrete artefact derived from `ItemScheme` such as a `CodeList`, `ConceptScheme`, `CategoryScheme`, or `OrganisationScheme`). For instance if the `targetClass` of the `IdentifierComponent` is a `DataProvider` then the specialisation of (i.e. type of) *`ItemScheme`* will be an `OrganisationScheme` containing a list of `DataProviders`. Normally, such an *`ItemScheme`* can be specified in the `MetadataStructureDefinition`. However, there will be cases where this is not possible. An example of this where the `IdentifierComponent` is a `Dimension` in a `KeyFamily` – as individual `Dimensions` can use `Concepts` from different `ConceptSchemes` it is necessary for an application to read the `KeyFamily` definition in order to validate that a correct `Concept` is referenced in the `IdentifierComponentValue` of the `MetadataSet`.

The `PartialTargetIdentifier` identifies a sub set of the `IdentifierComponents` of the `FullTargetIdentifier`. The purpose here is to ensure that a single `MetadataStructureDefinition` can be defined for a discrete set of related object types: thus, for example, a single definition can be constructed to define the metadata that can be attached to any part of a key family, or that can be attached to any artefact concerned with the reporting of quality metadata (such as data provider and (data)category). The `FullTargetIdentifier` will identify all the relevant object types that are in the scope of the definition, whilst the `PartialTargetIdentifier` will identify a sub set of these object types which form the “key” of `targetClass` of the

PartialTargetIdentifier. For example, in a key family the targetClass might be a dimension, and therefore the IdentifierComponents are those that uniquely identify a dimension (which, incidentally, are the key family, and the concept).

The ReportStructure comprises a set of *MetadataAttribute*s that can be defined as a hierarchy. Each *MetadataAttribute* identifies a Concept that is reported or disseminated in a MetadataSet that uses this MetadataStructureDefinition. The Concept must be a valid Concept maintained in a ConceptScheme. It is not mandatory that all *MetadataAttribute*s are linked to Concepts from the same ConceptScheme.

The *MetadataAttribute* can be specified as being mandatory, conditional, or optional (assignmentStatus – inherited from *Attribute*).

The *MetadataAttribute* is an abstract class and is either a CodedMetadataAttribute or an UncodedMetadataAttribute. A CodedMetadataAttribute is associated to the CodeList that contains the set of valid values that can be reported for the CodedMetadataAttribute in a MetadataSet.

It is possible to define a sub structure of the *MetadataAttribute* by use of the AttributeProperty.

The AttributeProperty allows the *MetadataAttribute* to have identifiable text (such as a URL). However, there is no support for sequencing and applications must know how to integrate the value of the property sent in a MetadataSet with any value sent in the body of the UncodedMetadataAttribute or CodedMetadataAttribute.

Each *MetadataAttribute* can be specified as being attachable to one or more *IdentifiableArtefact*. The diagram below shows the classes that inherit from *IdentifiableArtefact* in the context of reference metadata.

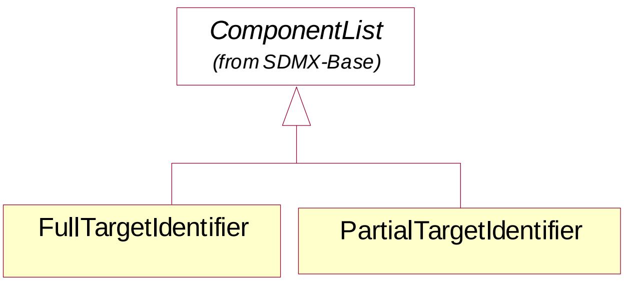


Figure 30: Metadata Attribute attachment definition

It can be seen from this that the specification of the object types to which a *MetadataAttribute* can be attached is indirect: the *MetadataAttribute* is attached to one or more of FullTargetIdentifier or PartialTargetIdentifier and the actual object is identified by the targetClass to which the FullTargetIdentifier or

PartialTargetIdentifier is associated. This gives a flexible mechanism by which the actual object types need not be defined in concrete terms in the model, but are defined dynamically in the MetadataStructureDefinition, in much the same way as the keys to which data observation are "attached" in a KeyFamily definition. In this way the MetadataStructureDefinition can be used to define any set of *MetadataAttributes* and any set of object types to which they can be attached.

Each *MetadataAttribute* can have a Type and *Representation* specified (using the localType and localRepresentation associations). If this is not specified in the MetadataStructureDefinition then the Type and *Representation* is taken from that defined for the Concept (the coreType and coreRepresentation associations).

The definition of the various types of of *Representation* and the Type can be found in section 4.4.

The MetadataStructureDefinition is linked to a MetadataflowDefinition. The MetadataflowDefinition does not have any specific attributes but can have additional metadata attached using the reference metadata mechanism itself.

Of importance is the fact that the MetadataflowDefinition associates a MetadataStructureDefinition with one or more Category (possibly from different CategorySchemes). This gives a system the ability to state which MetadataSets are to be reported/disseminated for a given Category, and which MetadataSets can be reported using the MetadataStructureDefinition.

7.3.4.2 Definitions

Class	Feature	Description
StructureUsage		See "SDMX Base".
	classify	Associates one or more Categories in one or more schemes that define metadata data categorisation in terms of metadata to be reported or disseminated.
Category		See "Category Scheme".
Metadataflow Definition	Inherits from: *StructureUsage*	Abstract concept (i.e. the structure without any metadata) of a flow of metadata that providers will provide for different reference periods.

Class	Feature	Description
	/structure	Associates a Metadata Structure Definition.
MetadataStructure Definition		A collection of metadata concepts, their structure and usage when used to collect or disseminate reference metadata.
	/grouping	An association to a set of metadata concepts that have an identified structural role in a Metadata Structure Definition.
FullTarget Identifier	Inherits from ComponentList	A set components that define the key of an object type to which metadata may be attached.
	/components	Associates the Identifier Components that define the key.
	targetClass	An association to the Identifiable Object Type that the Target Identifier identifies.
PartialTarget Identifier	Inherits from ComponentList	A set components that define a key of an object type to which metadata may be attached, and which is a partial key of the object identified in the Full Target Identifier.
	/components	Associates the Identifier Components that defines the partial key
	targetClass	An association to the Identifiable Object Type that the Partial Target Identifier identifies.
IdentifierComponent		A Concept used to refer to and identify a part of an identifier in a Metadata Structure Definition.
	targetClass	An association to the Identifiable Object Type that the Identifier Component identifies.
	codelist	Associates an Item Scheme such as a Code List, Concept Scheme, and Category Scheme.

Class	Feature	Description
ItemScheme	Sub classes: CodeList ConceptScheme CategoryScheme OrganisationScheme	The list of values that defines the value domain of the Identifier Component.
ConceptDescriptor	Inherits from: *ComponentList*	A set metadata concepts that define the metadata attributes of a Metadata Structure Definition
	/components	An association to the Metadata Attributes relevant to the Metadata Structure Definition.
MetadataAttribute	Abstract class Sub classes are: *CodedMetadataAttribute* *UncodedMetadataAttribute*	The value of an attribute, such as the instance of a coded or uncoded attribute in a Metadata Structure Definition.
	/conceptIdentity	An association to the metadata concept which defines the semantic of the attribute.
	properties	Allows one or more Attribute Property to be defined as a sub structure of the MetadataAttribute.
	/localType	Associates a Type (data type) that overrides any core type specified for the Concept itself.
	/localRepresentation	Associates a Representation that overrides any core representation specified for the Concept itself.
Concept	Inherits from: Item	The metadata concept which defines the semantic of the Metadata Attribute in the Metadata Structure Definition
AttributeProperty		A specific characteristic of a structure identified by its name and type.
	name	The name of the Attribute Property

Class	Feature	Description
	type	Specifies the data type for the Attribute Property. The types are an enumerated list in the Data Type enumeration.
IdentifiableArtefact		Specifies to which artefacts the Metadata Attribute can be attached. This is constrained to the Full Target Identifier or the Partial Target Identifier.
CodedMetadataAttribute	Inherits from *MetadataAttribute* *CodedArtefact*	A Metadata Attribute that takes its values from a code list.
	/codelist	Associates a Code List.
UncodedAttribute	Inherits from *MetadataAttribute* *UncodedArtefact*	A metadata attribute whose content is uncoded.

7.4 Metadata Set

7.4.1 Class Diagram

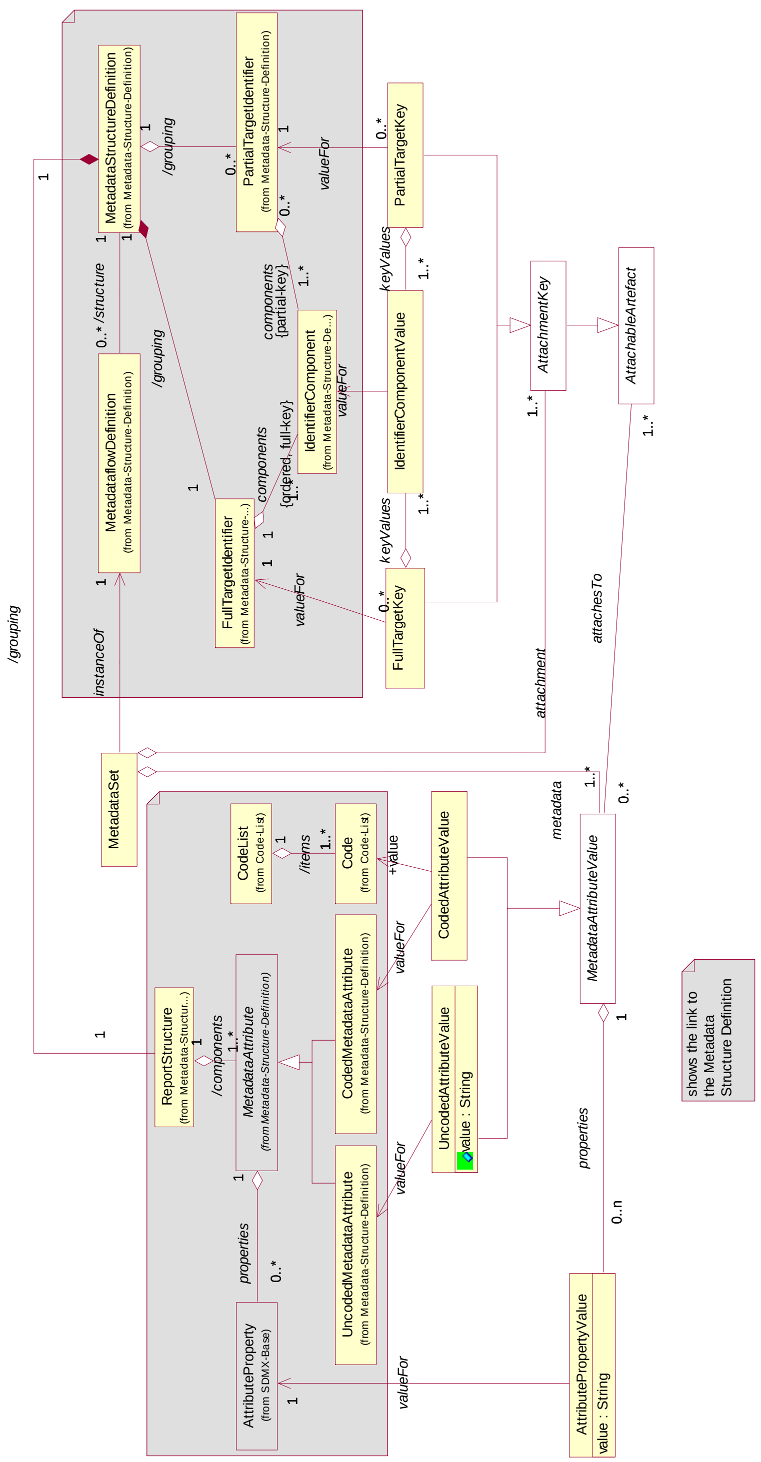


Figure 31: The Metadata Set

7.4.2 Explanation of the Diagram

7.4.2.1 Narrative

The classes in the shaded boxes on the class diagram comprise the classes in the MetadataStructureDefinition. They are included in this diagram to show the link between the contents of the MetadataSet and the structures in the MetadataStructureDefinition. Depending on implementation architectures, it is possible to include just a reference to the MetadataflowDefinition in an instance of the MetadataSet (as the MetadataflowDefinition uses just one MetadataStructureDefinition).

A MetadataSet comprises a set of MetadataAttributeValues that give additional meaning to the object identified by the FullTargetKey or PartialTargetKey. The component structure of the key is specified in the FullTargetIdentifier or PartialTargetIdentifier defined in the MetadataStructureDefinition.

The set of IdentifierComponentValue for the TargetIdentifier is defined in the TargetKey, and for the PartialTargetIdentifier these are defined in the PartialTargetKey.

The MetadataSet contains *MetadataAttributeValue*s, each of which is associated to (attached to) an *AttachableArtefact*. The *AttachmentKey* is a specialisation of *AttachableArtefact* which has, as concrete classes, the FullargetKey and the PatialTargetKey. Therefore a *MetadataAttributeValue* can be attached to one or both of the FullTargetKey and PartialTargetKey. A simple text value for the attribute uses the UncodedAttributeValue sub class of *MetadataAttributeValue* whilst a coded value uses the CodedAttributeValue sub class.

The metadata reported for a *MetadataAttributeValue* may additionally have one or more AttributePropertyValues, if the AttributeProperty has been specified as being allowed for the *MetadataAttribute* in the MetadataStructureDefinition.

7.4.2.2 Definitions

Class	Feature	Description
MetadataSet		Any organised collection of metadata.
	effectiveDate	The date on which all the metadata in the metadata set is effective.
	instanceOf	Associates the MetadataflowDefinition for which this Metadata Set is an instance.
	attachmentKey	Associates the object keys to which metadata is to be attached.

Class	Feature	Description
	metadata	Associates the Metadata Attribute Values which are to be associated with the object or objects identified by a key.
AttachableArtefact	Abstract class Sub class: AttachmentKey	Links to the object to which the metadata are to be attached.
AttachmentKey	Abstract class Sub classes are: TargetKey PartialTargetKey	Identifies the key of the object to which the metadata are to be attached.
FullTargetKey	Inherits from AttachmentKey	The key of an individual object of the type specified in the Full Target Identifier of the Metadata Structure Definition.
	keyValues	Associates the identifier component values of the Target Identifier.
	valueFor	Associates the target identifier that identifies the object type and the component structure of the key.
PartialTargetKey	Inherits from AttachmentKey	The key of an individual object of the type specified in the Partial Target Identifier of the Metadata Structure Definition.
	valueFor	Associates the Partial Target Identifier that identifies the object type and the component structure of the Partial Target Key.
	keyValues	Associates the Identifier Component values of the Target Identifier.

Class	Feature	Description
IdentifierComponent Value		The value of an individual component of the Target Identifier or Partial Target Identifier. The concatenation of the identifier values comprises the key of an individual object.
MetadataAttribute Value	Abstract class Sub classes are: *UncodedAttributeValue* *CodedAttributeValue*	The value for a Metadata Attribute
	valueFor	Association to the Metadata Attribute in the Metadata Structure Definition that identifies the Concept, Code List, properties, and data type of the attribute.
	properties	Association to one or more Property Values.
	attachesTo	Association to the attachable artefact (i.e. the Target Key or Partial Target Key) to which the Metadata Attribute Value pertains.
AttributeProperty Value		The value of a property which gives additional metadata for the Metadata Attribute Value.
	value	The content of the property metadata.
	valueFor	Association to the Property for the Metadata Attribute in the Metadata Structure Definition that identifies the name and type of the property value.
UncodedAttribute Value	Inherits from *MetadataAttributeValue* Sub class: *XMLAttributeValue*	The text content of an attribute.
CodedAttributeValue	Inherits from *MetadataAttributeValue*	The coded content of an attribute.

Class	Feature	Description
	+value	Association to a Code in the Code List that is the value of the attribute.

1563

8 HIERARCHICAL CODE SCHEME

8.1 Scope

The `CodeList` described in the section on structural definitions supports a simple hierarchy of `Codes`, and restricts any child `Code` to having just one parent `Code`. Whilst this structure is useful for supporting the needs of the `KeyFamily` and the `MetadataStructureDefinition`, it is not sufficient for supporting the more complex associations between codes that are often found in coding schemes such as a classification scheme. Often, the `CodeList` used in a `KeyFamily` is derived from a more complex coding scheme. Access to such a coding scheme can aid applications, such as OLAP applications, to give more views of the data than would be possible with the simple `CodeList` used in the `KeyFamily`.

Note that a hierarchical code list is not necessarily a balanced tree. A balanced tree is where levels are pre-defined and fixed, (i.e. a level always has the same set of codes, and any code has a fixed parent and child relationship to other codes). A statistical classification is an example of a balanced tree, and the support for a balanced hierarchy is a sub set, and special case, of the hierarchical code list.

The principle features of the Hierarchical Code Scheme are:

1. A child code can have more than one parent.

2. There can be more than one code that has no parent (i.e. more than one “root node”).

3. There may be many hierarchies (or “views”) defined, in terms of the associations between the codes. Each hierarchy serves a particular purpose in the reporting, analysis, or dissemination of data.

8.2 Inheritance

8.2.1 Class Diagram

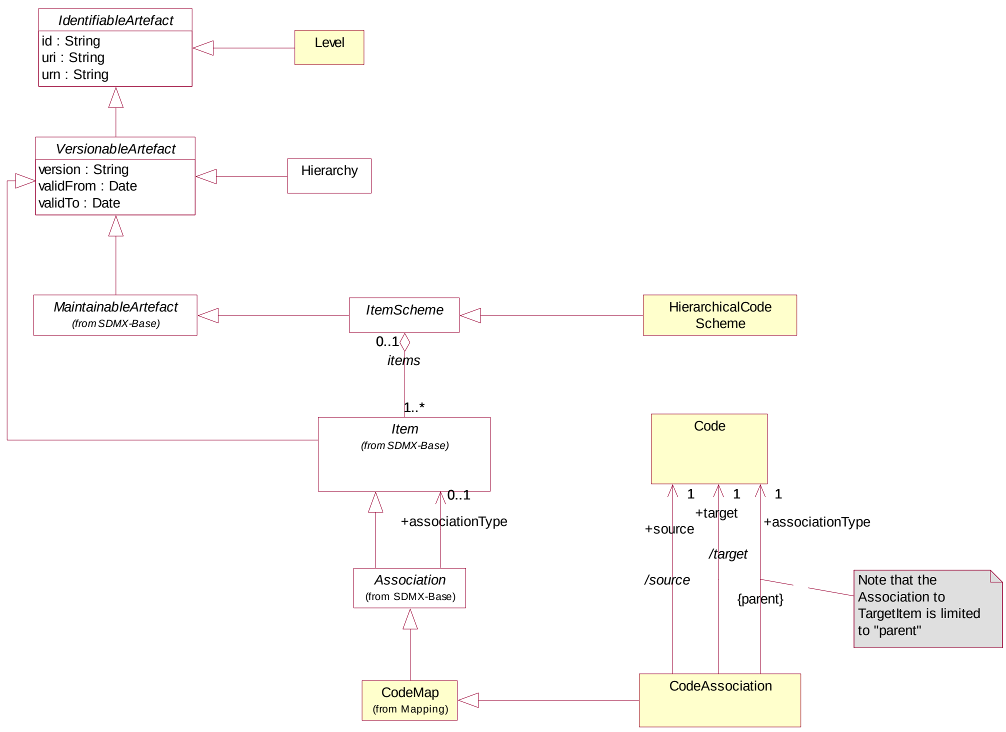


Figure 32: Inheritance class diagram for the Code Set

8.2.2 Explanation of the Diagram

8.2.2.1 Narrative

[General note: The constraints on the inherited associations (e.g. between `CodeAssociation` and `Code`) are shown in the context of the functionality of the `HierarchicalCodeScheme`. This does not mean that other association roles cannot be placed on a `Code` participating in a `HierarchicalCodeScheme` (such as may be defined in a CodeMap – see section 9. The class diagram merely restricts or constrains the associations to that usage required to support the functionality of the `HierarchicalCodeScheme`.]

The `HierarchicalCodeScheme` inherits from *`ItemScheme`* and is therefore a *`MaintainableArtefact`* with identification, versioning and a maintenance agency. The `CodeAssociation` inherits from `CodeMap` (see section 9) and is therefore a *`VersionableArtefact`*. *`Hierarchy`* inherits directly from *`VersionableArtefact`*. These two therefore have identity and versioning. The `Level` is an *`IdentifiableArtefact`* and therefore has an Id, multi-lingual name and multi-lingual description.

It is important to understand that the `Codes` participating in a `HierarchicalCodeScheme` are not themselves contained in the scheme – they are referenced from the scheme and are maintained in one or more `CodeLists`. This is explained in the explanation of the relationship class diagram below.

The associations between `CodeAssociation` and `Code` are inherited from the associations between `CodeMap` and `Code`. However, the derived associations are constrained further as follows:

- The association defining the relationship between the source and target codes is restricted to the “parent” relationship (i.e. the target `Code` is the parent)

Note that the `Code` associated by the `associationType` is not in the same `CodeList` as either the source or target code – it is in a specific `CodeList` of role types.

8.2.2.2 Definitions

The definitions of the various classes, attributes, and associations are shown in the relationship section below.

8.3 Relationship

8.3.1 Class Diagram

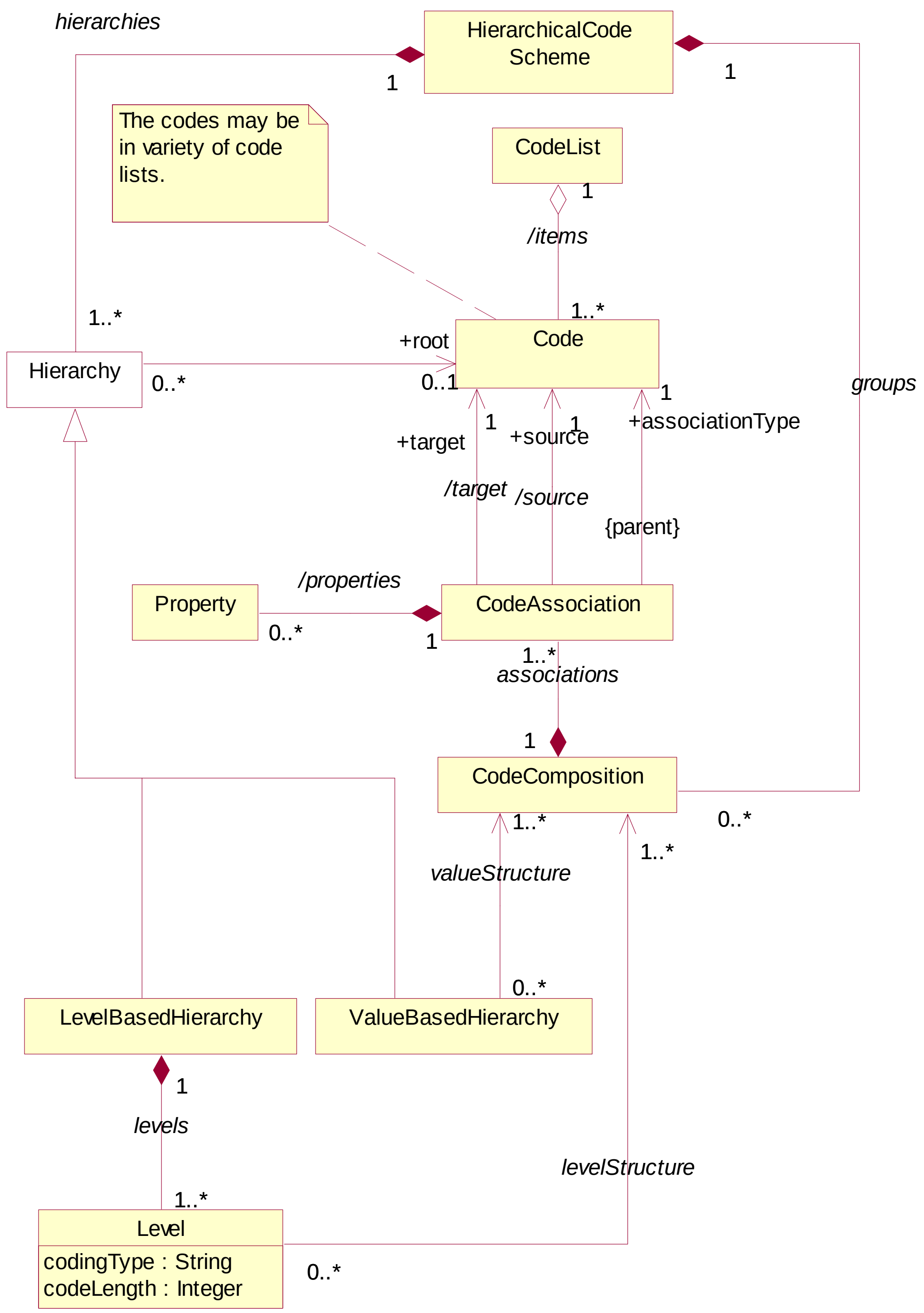


Figure 33: Relationship class diagram of the Hierarchical Code Scheme

8.3.2 Explanation of the Diagram

8.3.2.1 Narrative

The associations and navigability of the associations in the HierarchicalCodeScheme is constrained in such a way so as to ensure a consistent common implementation of the HierarchicalCodeScheme in terms of basic functionality. Of key importance are:

1. The HierarchicalCodeScheme is a specification of the Codes comprising the scheme and the specification of the structure of the Codes in the scheme in terms of one or more *Hierarchy.*

2. The Codes in the HierarchicalCodeScheme are not themselves a part of the scheme, rather they are references to Codes in one or more external CodeLists.

3. These Codes may participate in one or more Hierarchy, and one or more CodeComposition in order to give structure to the HierarchicalCodeScheme.

4. The association between any two codes is specified in a CodeAssociation. The association is limited to identifying a Code and its parent Code.

5. The parent Code is the same for all CodeAssociations comprising a CodeComposition.

Relationships

Relationships between the codes are defined in the CodeComposition, which itself comprises a number of CodeAssociations. The CodeAssociation links a Code (source) to a parent Code (target). The constraint is that the parent code in each of the CodeAssociations of the CodeComposition must be the same Code. The CodeAssociation can have one or more Property which allow the definition of properties, e.g. a sequence number or the relative weight of a (child) Code in respect to its parent's decomposition.

A Code can participate in one or more CodeAssociation, playing the role of source (child) or target (parent). A Code can play both roles but in different CodeAssociation linked to different CodeCompositions.

Hierarchies

It is possible to define formal hierarchies of Codes, and a HierarchicalCodeScheme may have more than one such *Hierarchy*. Each *Hierarchy* can identify the root Code. There are two types of *Hierarchy* – value based and level based.

A ValueBasedHierarchy comprises a set of CodeComposition (any combination is allowable in principle).

A `LevelBasedHierarchy` supports the need where formal levels need to be defined. Each `Level` comprises a set of `CodeComposition`. The constraint of a `LevelBasedHierarchy` is that each `Code` in a `Level` has one and only one parent in the superior `Level`. Note that statistical classifications are often structured as a `LevelBasedHierarchy`.

The `Level` inherits from *`IdentifiableArtefact`* and therefore has an `Id`, multi-lingual `name`, multi-lingual `description`, and `Annotation`.

[Note that organisations wishing to be compliant with accepted models for statistical classifications should ensure that the `Id` is the number associated with the `Level`, where `Levels` are numbered consecutively starting with level `1` at the highest `Level`].

The `ItemProperty` allows one or more optional properties to be defined for the `CodeAssociation`. In the context of the `HierarchicalCodeScheme`, a property could be the sequence in which the source code participates in the `CodeComposition`.

8.3.2.2 Definitions

Class	Feature	Description
HierarchicalCodeScheme	Inherits from: ItemScheme	An organised collection of codes that may participate in many parent/child relationships with other Codes in the scheme, as defined by one or more Hierarchy of the scheme.
	groups	Association to groupings of Codes.
	hierarchies	Association to Hierarchies of Codes.
CodeComposition		A group of Codes where all Codes in the group have an association with the same parent Code.
	associations	Association to an association of two Codes.
CodeAssociation	Inherits from CodeSet	An association between two Codes.
	+source	Association to the source Code
	+target	Association to the target Code.

Class	Feature	Description
	+associationType	The role of the association between source and target Code. This is constrained to "parent" (i.e. the target Code is the parent Code).
Code		The source or target Code
	/items	Association to the Code List containing the Code.
CodeList		The Code List containing te Code.
Hierarchy	Abstract class Sub classes are: LevelBasedHierarchy ValueBasedHierarchy	A classification structure arranged in levels of detail from the broadest to the most detailed level.
	+root	Association to the top level code in the hierarchy.
LevelbasedHierarchy	Inherits from Hierarchy	A hierarchy structure where the structure is arranged in levels of detail from the broadest to the most detailed level. Each level is defined in terms of the categories at the next lower level of the hierarchy.
	levels	Association to the levels in the hierarchy.
Level		A group of Codes which are characterised by homogeneous coding, and where the parent of each Code in the group is at the same higher level of the Hierarchy.
	codingType	Indicates whether the codes at the level are alphabetical, numerical or alphanumerical
	codeLength	Number of characters which the codes at this level have.
	levelStructure	Association to the code groups comprising the level.

Class	Feature	Description
`ValueBasedHierarchy`	Inherits from `Hierarchy`	A hierarchy structure where the items in the hierarchy have no formal level structure.
	`valueStructure`	Association to the code groups comprising the Hierarchy.

1705

9 STRUCTURE SET AND MAPPINGS

9.1 Scope

A `StructureSet` allows components in one structure to be mapped to components in another structure of the same type. In this context the term "structure" is used loosely to include types of *`ItemScheme`*, types of *`Structure`*, and types of *`StructureUsage`*. The allowable structures that can be mapped, and the components that can be mapped within these structures are:

Structure Type	Component type
Code List	Code
Category Scheme	Category
Concept Scheme	Concept
Data Structure Definition (Key Family)	Dimension, Data Attribute, Measure
Metadata Structure Definition	Identifier Component, Metadata Attribute
Dataflow Definition	Data Structure Definition (Key Family)
Metadataflow Definition	Metadata Structure Definition

The `StructureSet` can contain one or more "maps" and can define a hierarchy of maps which effectively group relevant sub component maps. An example of this is:

Dataflow Definition ➔ Data Structure Definition ➔ [Dimension, Data Attribute, Measure] ➔ Code List ➔ Code.

9.2 Structure Set

9.2.1 Class Diagram

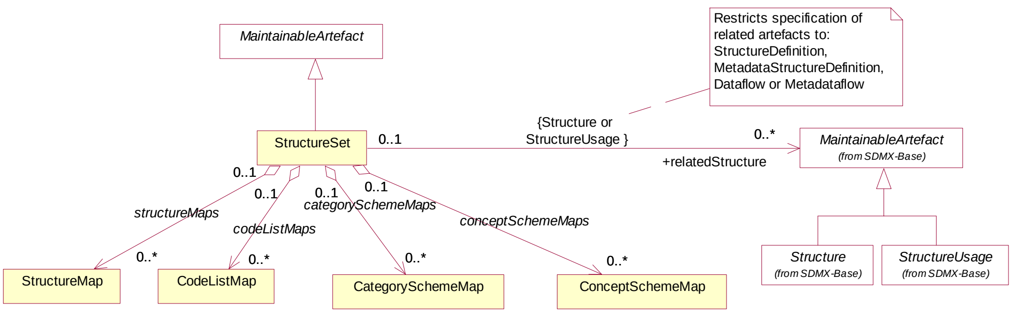


Figure 34: Class diagram of the Structure Set

9.2.2 Explanation of the Diagram

9.2.2.1 Narrative

The `StructureSet` is a *`MaintainableArtefact`*. It can contain:

1. A set of references to concrete sub-classes of *`Structure`* and *`StructureUsage`* (`KeyFamily`, `MetadataStructureDefinition`, `DataflowDefinition` or `MetadataflowDefinition`) to indicate that a semantic relationship exist between them. For example there may be group of `KeyFamily` which, together, form the definition of a cube, each `KeyFamily` defining a part of the cube.

2. A set of `StructureMaps` which define which components of one structure are equivalent to those in another.
3. A set of `CodeListMaps` which define how `Codes` are mapped between `CodeLists` or *`Hierarchy`*.
4. A set of `CategorySchemeMaps` which define how `Categorys` are mapped between `CategorySchemes`.
5. A set of `ConceptSchemeMaps` which define how `Conceptss` are mapped between `ConceptSchemes`.

9.2.2.2 Definitions

Class	Feature	Description
StructureSet		A maintainable collection of structural maps that link components together in a source/target relationship where there is a semantic equivalence between the source and the target components.
	+relatedStructure	Association to one of: Key Family (Data Structure Definition); Metadata Structure Definition; Dataflow Definition; Metadataflow Definition.
	structureMaps	Association to Structure Maps.
	codeListMaps	Association Code List Maps.
	categorySchemeMaps	Association to a Category Scheme Map.
	conceptSchemeMaps	Association to Concept Scheme Maps.

9.3 Structure Map

9.3.1 Class Diagram

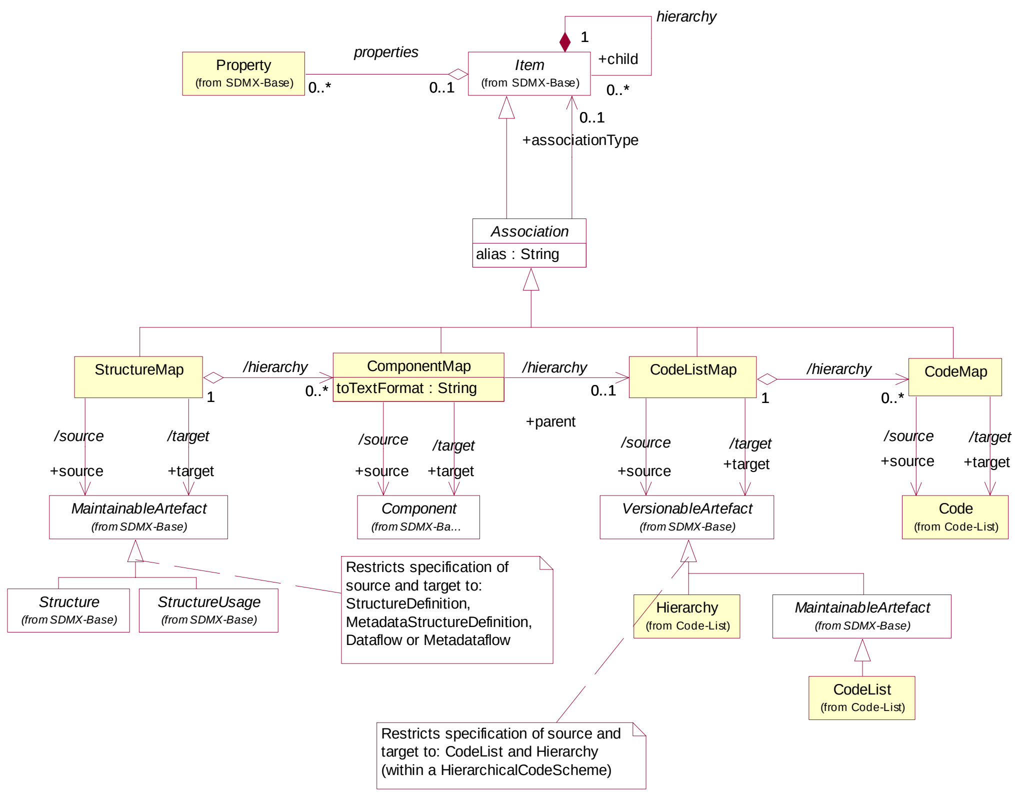


Figure 35: Class diagram of the Structure Map

9.3.2 Explanation of the Diagram

9.3.2.1 Narrative

The `StructureMap` references two *`Structures`* or *`StructureUsages`*. In concrete terms these references will be to `DataStructureDefinitions`, `MetadataStructureDefinitions`, `DataflowDefinitions` or `MetadataflowDefinitions`. The `StructureMap` contains a set of `ComponentMaps`, each one indicating equivalence between Components of the referenced *`Structure`* or *`StructureUsage`*. `ComponentMap` has the attribute `toTextFormat` which takes values: `id`, `name`, `description`. This instructs mapping tools to use the id, name or description of a coded component to determine equivalence with an uncoded component's value. For each indicated *`Component`* equivalence (this is effectively `Concept` equivalence), a `CodeListMap` may be defined.

An example of a `ComponentMap` is linking the source *`Component`* that is a `Dimension` in the source `KeyFamily` (identified in the `StructureMap`) to the equivalent target *`Component`* that is a `Dimension` in the target `KeyFamily`).

The `CodeListMap` references two `CodeLists` or `Hierarchy` (within a `HierarchicalCodeScheme`). The `CodeListMap` contains a set of `CodeMaps`, each one indicating equivalence between `Codes` of the referenced `CodeLists`. Again, the alias attribute can provide a name for all equivalent codes in multiple "pair-wise-joined" `CodeLists` to facilitate querying. The `CodeListMap` can either be hierachically linked to the `ComponentMap` or it can be specified independent of a `ComponentMap`.

Each of the maps inherits from *`Association`* and therefore inherits the association to `Property`, thus allowing additional properties to be defined for the map.

9.3.2.2 Definitions

Class	**Feature**	**Description**
`StructureMap`	Inherits from *`Association`*	Links a source and target structure where there is a semantic equivalence between the source and the target structures.
	`+source`	Association to the source structure.
	`+target`	Association to the target structure.
	`/hierarchy`	Association to the Component Maps.
`ComponentMap`		Links a source and target Component where there is a semantic equivalence between the source and the target Components.
	`+source`	Association to the source Component.
	`+target`	Association to the target Component.
	`/hierarchy`	Association to the Code List Maps.
`CodeListMap`		Links a source Code List or Hierarchy to a target Code List or Hierarchy where there is a semantic equivalence between the source and the target Code List or Hierarchy.
	`+source`	Association to the source Code List or Hierarchy.
	`+target`	Association to the target Code List or Hierarchy.
	`/hierarchy`	Association to the Code Maps.

Class	Feature	Description
`CodeMap`		Links a source and target Code where there is a semantic equivalence between the source and the target Codes.
	`+source`	Association to the source Code.
	`+target`	Association to the target Code.

9.4 Concept Scheme Map and Category Scheme Map

9.4.1 Class Diagram

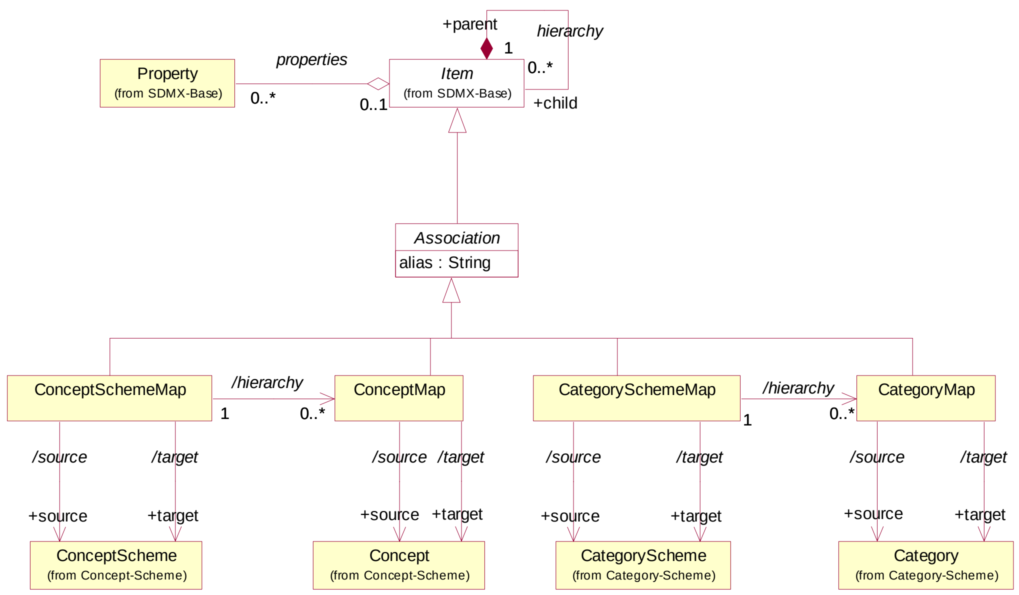


Figure 36: Class diagram of the Concept Scheme Map and Category Scheme Map

9.4.2 Explanation of the Diagram

9.4.2.1 Narrative

The `ConceptSchemeMap` provides a mechanism for specifying semantic equivalence between Concepts. It identifies two `ConceptSchemes` whose `Concepts` are to be mapped. Note that many schemes can be joined together via a set of pair-wise mappings. The `ConceptMap` denotes which `Concepts` are semantically equivalent and an alias can be specified to refer to a set of mapped concepts to facilitate querying.

The `CategorySchemeMap` is analogous to the `ConceptSchemeMap`, except that its use is targeted towards expressing semantic equivalence in `CategorySchemes` such as a subject-matter domain scheme.

1792 **9.4.2.2 Definitions**

Class	Feature	Description
ConceptSchemeMap		Links a source and target Concept Scheme where there is a semantic equivalence between the source and the target schemes.
	+source	Association to the source Concept Scheme.
	+target	Association to the target Concept Scheme.
	/hierarchy	Association to the Concept Maps.
Concept Map		Links a source and target Concept where there is a semantic equivalence between the source and the target Concepts.
	+source	Association to the source Concept.
	+target	Association to the target Concept.
CategorySchemeMap		Links a source and target Category Scheme where there is a semantic equivalence between the source and the target schemes.
	+source	Association to the source Category Scheme.
	+target	Association to the target Category Scheme.
	/hierarchy	Association to the Category Maps.
Concept Map		Links a source and target Category where there is a semantic equivalence between the source and the target Category.
	+source	Association to the source Category.
	+target	Association to the target Category

1793

10 DATA CONTRAINTS AND PROVISIONING

10.1 Scope

The scope of this section is to describe the support in the metamodel for specifying both the access to and the content of a data source. The information may be stored in a resource such as a registry for use by applications wishing to locate data and metadata which is available via the Internet.

Note that in this metamodel the term data source refers to both data and metadata sources, and data provider refers to both data and metadata providers.

A data source may be a simple file of data or metadata (in SDMX-ML format), or a database or metadata repository. A data source may contain data for many data or metadataflows (called `DataflowDefinition`, `CubeDefinition`, and `MetadataflowDefinition` in the model), and the mechanisms described in this section allow the `DataProvider` to specify precisely the scope of the content of the data source.

The `DataflowDefinition`, `MetadataflowDefinition`, and `CubeDefinition` themselves may be specified as containing only a sub set of all the possible keys that could be derived from a `KeyFamily`, `MetadataStructureDefinition`, or `CubeStructure`. A `DataProvider` may further constrain this set of keys by describing the sub set that is available in the data or metadata source. These specifications are called *`Constraint`* in this model.

10.2 Inheritance

10.2.1 Inheritance Class Diagram of Constrainable and Data Provisioning Artefacts

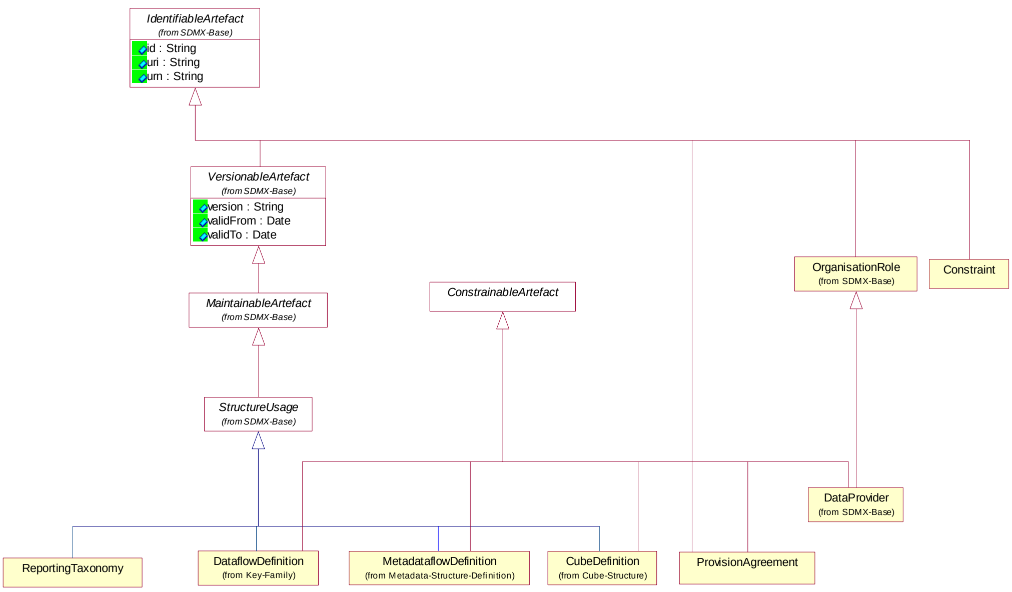


Figure 37: Inheritance class diagram of constrainable and provisioning artefacts

10.2.2 Explanation of the Diagram

10.2.2.1 Narrative

Any artefact that is derived from `ConstrainableArtefact` can have constraints defined. The artefacts that can have constraint metadata attached are:

- `DataflowDefinition`
- `ProvisionAgreement`
- `DataProvider`
- `MetadataflowDefinition`
- `CubeDefinition`

10.3 Constraints

10.3.1 Relationship class diagram of constraint metadata

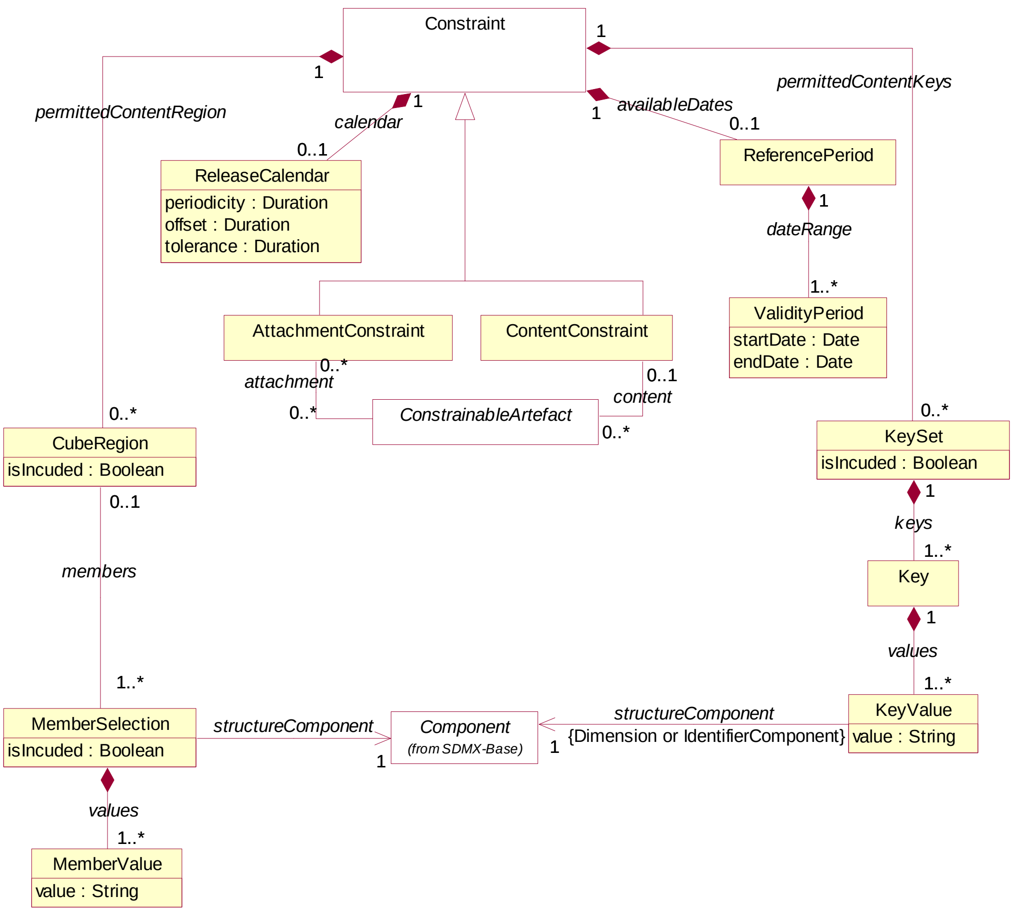


Figure 38: Relationship class diagram showing constraint metadata

10.3.2 Explanation of the Diagram

10.3.2.1 Narrative

The constraint mechanism allows specific constraints to be attached to a *ConstrainableArtefact*. With the exception of ReleaseCalendar, these constraints specify a sub set of the total set of values or keys that may be present in a DataSet or MetadataSet. The total set of values are those that can be inferred from the relevant structure definition (KeyFamily, MetadataStructureDefinition, and CubeStructure).

For instance a KeyFamily specifies, for each Dimension, the list of allowable code values. However, a specific DataflowDefinition that uses the KeyFamily may contain only a sub set of the possible range of keys that is theoretically possible from the KeyFamily definition (the total range of possibilities is sometimes called the cartesian product of the dimension values). In addition to this, a DataProvider that is capable of supplying data according to the DataflowDefinition has a ProvisionAgreement, and the DataProvider may also wish to supply constraint metadata which may further constrain the range of possibilities in order to describe the data that the provider can supply.

A *ConstrainableArtefact* can have two types of *Constraint*:

1. ContentConstraint – is used solely as a mechanism to specify either the available set of keys (KeySet) or set component values (CubeRegion) in a *DataSource* such as a DataSet or a database (*QueryDatasource*). Only one such constraint may be present for a *ConstrainableArtefact*.

2. AttachmentConstraint – is used as a mechanism to define slices of the full set of data and to which other object types in the model (such as a CubeComponent – see **Error! Reference source not found.**) may be attached. These slices can be defined either as a set of keys (KeySet) or a set component values (CubeRegion). There can be many AttachmentConstraints specified for a specific AttachableArtefact.

A *Constraint* is an *IdentifiableArtefact* and can therefore be associated with one or more *AttachableArtefacts*. However, because the Constraint can specify a sub set of the component values implied a specific *Structure* (such a specific KeyFamily or specific CubeStructure) then all of the *AttachableArtefacts* must be associated with the same specific *Structure*.

A *Constraint* has a choice of two ways of specifying value sub sets:

1. As a set of keys (KeySet) that can be present in the DataSet or MetadataSet. The KeySet specifies a number of Keys in terms of their KeyValues. Each KeyValue is a value that may be present for a *Component* (specifically a Dimension or IdentifierComponent) of a structure when contained in a DataSet or MetadataSet.

2. As a set of CubeRegions each of which defines a "slice" of the total structure in terms of one or more values that may be present for a *Component* (which

can be any type of *`Component`*) of a structure when contained in a `DataSet` or `MetadataSet.`

The difference between (1) and (2) above is that in (1) a complete key is defined whereas in (2) above a `CubeRegion` defines a list of possible values for each of the *`Components`* but does not specify specific key combinations. In addition, in (1) the association between *`Component`* and `KeyValue` is constrained to the components that comprise the key or identifier, whereas in (2) it can contain other component types (such as attributes). The value in `KeyValue.value` and `MemberValue.value` must be consistent with the *`Representation`* declared for the *`Component`* in the `KeyFamily` or the `MetadataStructureDefinition` linked to the `DataflowDefintion` or `MetadataflowDefinition.` Note that in all cases the “operator” on the `value` is deemed to be “equals”.

It is possible to define for the `KeySet, CubeRegion`, and `MemberSelection` whether the set is included (`isIncluded` = “true”) or excluded (`isIncluded`=”false”) from the constraint definition. This attribute is useful if, for example, only a small sub-set of the possible values are not included in the set, then this smaller sub-set can be defined and excluded from the constraint.

In addition to `KeySets` or `CubeRegions`, a `Constraint` can have:

- a `ReferencePeriod` defining one of more date ranges (`ValidityPeriod`) specifying the time periods for which data or metadata are available

- a `ReleaseCalendar` that specifies the periodicity of the release of data or metadata

The `ReleaseCalendar` defines the planned release schedule in terms of periodicity and gives sufficient information to enable the calculation of a release schedule. The offset is calculated from the normal start date of the period as defined by ISO 8601 i.e. all periods start on the first relevant day on or after 1 January so a quarterly periodicity will have start periods of 1 January, 1 April, 1 July, and 1 October, and a weekly periodicity will start on the week that has first Thursday of the year.

10.3.2.2 Definitions

Class	Feature	Description

`Constrainable Artefact`	Abstract Class
	Sub classes are:
	
	`DataflowDefinition`
	`Metadataflow Definition`
	`CubeDefinition`
	`ProvisionAgreement`
	`DataProvider`
		An artefact that can have Constraints specified.

Class	Feature	Description
	content	Associates the metadata that constrains the content to be found in a data or metadata source linked to the Constrainable Artefact.
	attachment	Associates the metadata that constrains the valid content of a data or metadata set to which a Constrainable Artefact (such as Cube Item with the role "attribute") may be attached.
Constraint	Abstract class. Sub classes are: AttachmentConstraint ContentConstraint	Specifies a sub set of the definition of the allowable content of a data or metadata set in terms of the content or, for data only, in terms of the set of key combinations to which specific attributes (as defined by the Structure) may be attached.
	availableDates	Association to the set of time periods that identify the time ranges for which data are available in the data source.
	permittedContentKeys	Association to a sub set of Keys (i.e. value combinations) that can be derived from the definition of the Structure to which the Constrainable Artefact is linked.
	permittedContent Region	Association to a sub set of component values that can be derived from the definition of the Structure to which the Constrainable Artefact is linked.
	calendar	Association to a release calendar that defines dates on which the artefact is to be made available.

Class	Feature	Description
`ContentConstraint`	Inherits from *`Constraint`*	Defines a Constraint in terms of the content that can be found in data or metadata sets linked to the Constrainable Artefact to which this constraint is associated.
`Attachment Constraint`	Inherits from *`Constraint`*	Defines a Constraint in terms of the combination of component values that may be found in a data set, and to which a Constrainable Artefact may be associated in a structure definition.
`KeySet`		A set of keys.
	`isIncluded`	Indicates whether the Key Set is included in the constraint definition or excluded from the constraint definition.
	`keys`	Association to the keys.
`Key`		The set of Key Values comprising the Key.
	`values`	Associates the Key Values.
`KeyValue`		The value of a Component comprising a part of the Key.
	`structureComponent`	Association to the Component in the Structure to which the Constrainable Artefact is linked, which defines the valid Representation for the Key Value.
`Component`		See 3.5.3.2
`CubeRegion`		A set of Components and their values that defines a sub set or “slice” of the total range of possible content of the Structure to which the Constrainable Artefact is linked.

Class	Feature	Description
	isIncluded	Indicates whether the Cube Region is included in the constraint definition or excluded from the constraint definition.
	members	Associates the set of Components that define the sub set of values.
MemberSelection		A set of permissible values for one component of the axis.
	isIncluded	Indicates whether the Member Selection is included in the constraint definition or excluded from the constraint definition.
	structureComponent	Association to the Component in the Structure to which the Constrainable Artefact is linked, which defines the valid Representation for the Member Values.
MemberValue		The value of one Component of a Member Set.
	value	The value of the Component.
ReleaseCalendar		Defines the release schedule in terms of periodicity and timeliness.
	periodicity	The periodicity of the releases in terms of a known list of time periodicities (e.g. monthly, quarterly)
	offset	The offset in days from the normal start of the time period.
	tolerance	The number of days tolerance by which the release may be before or after the expected date.
ReferencePeriod		A set of dates that constrain the content that may be found in a data or metadata set.
	dateRange	Association to Validity Periods.

Class	Feature	Description
ValidityPeriod		A time period that defines a valid period.
	startDate	The start date of the period.
	endDate	The end date of the period.

10.4 Data Provisioning

10.4.1 Class Diagram

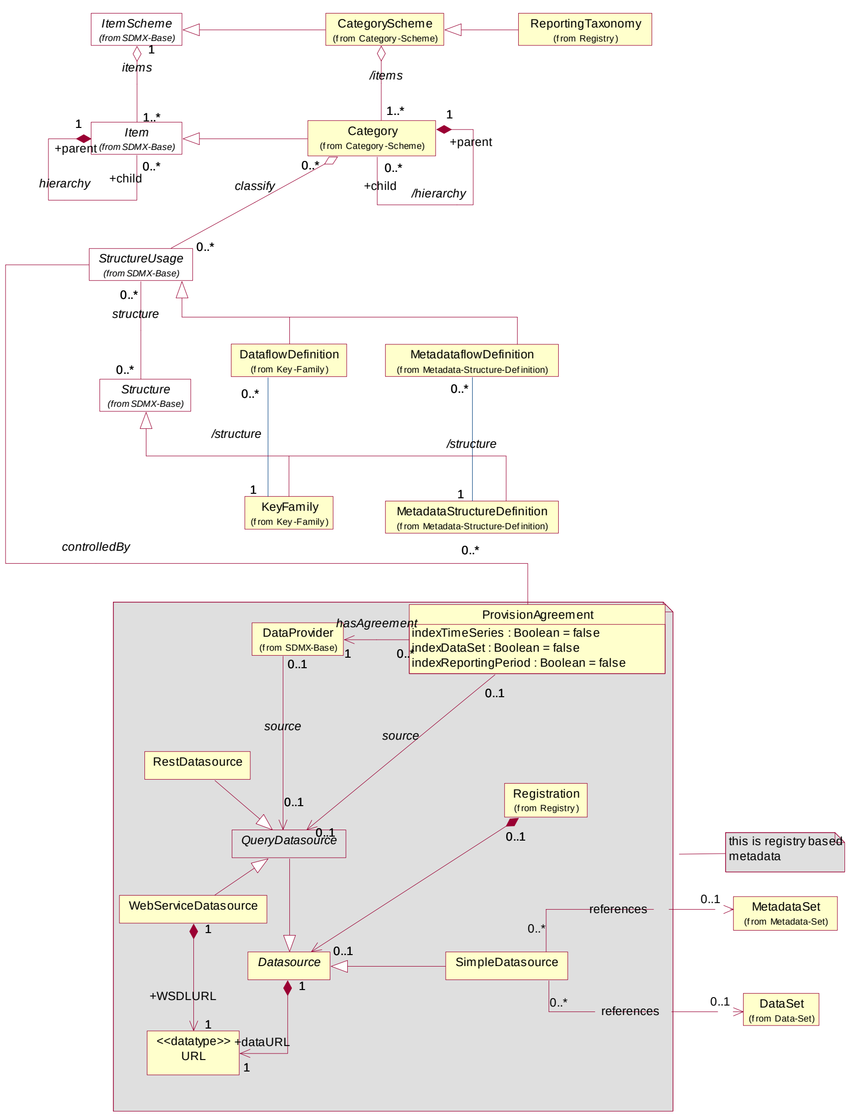


Figure 39: Relationship and inheritance class diagram of data provisioning

10.4.2 Explanation of the Diagram

10.4.2.1 Narrative

This sub model links many artefacts in the SDMX-IM and is pivotal to an SDMX metadata registry, as all of the artefacts in this sub model must be accessible to an application that is responsible for data and metadata registration or for an application that requires access to the data or metadata.

Whilst a registry can contain all of the metadata depicted on the diagram above, the classes in the grey shaded area are specific to a registry based scenario where data sources (either physical data and metadata sets or databases and metafdata repositories) are registered. More details on how these classes are used in a registry scenario can be found in the SDMX Registry Interface document.

A ProvisionAgreement links all the artefacts that define how data and metadata are structured and classified (*StructureUsage*) to the DataProvider, and it links to the Datasource, whether this be an SDMX conformant file on a website (SimpleDatasource) or a database service capable of supporting and SDMX query and responding with an SDMX conformant document (*QueryDatasource*).

The *StructureUsage*, which has concrete classes of DataflowDefinition, MetadataflowDefinition, and CubeDefinition identifies the corresponding KeyFamily, MetadataStructureDefinition, or CubeStructure, and it links to one or more Category in a CategoryScheme such as a subject matter domain scheme, by which the *StructureUsage* can be classified (for instance, to assist in drilling down from subject matter domains to find the data or metatata that may be relevant).

The ReportingTaxonomy allows an organisation to define a reporting scheme that defines many individual parcels of data, each structured differently, and combnines then in a reporting set. The ReportingTaxonomy itself has no detailed *Structure*, rather it has a high level structure defined in a CategoryScheme. Such schemes are common in primary reporting and this is described later (see 10.5).

The SimpleDatasource links to the actual DataSet or MetadataSet on a website (this is shown on the diagram as a dependency called "references"). The sourceURL is obtained during the registration process of the DataSet or the MetadataSet. The metadata about the content of the SimpleDatasource is stored in the registry in terms of a ContentConstraint (see 10.3) for the Registration.

The *QueryDatasource* links to the database or metadata repository that contains the data or metadata. The sourceURL is obtained during the registration process of the *QueryDatasource*. The metadata about the content of the *QueryDatasource* is stored in the registry in terms of a ContentConstraint (see 10.3) for the ProvisionAgreement or, in some cases, for the DataProvider. This later case is expected to be rare because even if the actual database is the same for all ProvisionAgreements for a DataProvider, it is probable that a specific ContentConstraint for the ProvisionAgreement gives more clarity to an application querying the registry about the relevance of the data source to fulfilling the specific scope of the query.

There are two types of *QueryDatasource*, the RestDatasource which is invoked using an HTTP “get”, and a WebServiceDatasource which conforms to a web service definition language (WSDL) profile that is available from the wsdURL.

10.4.2.2 Definitions

Class	Feature	Description
StructureUsage	Abstract class: Sub classes are: DataflowDefinition MetadataflowDefinition CubeDefinition ReportingTaxonomy	See 3.5.3.2
	controlledBy	Association to the Provision Agreements that comprise the metadata related to the provision of data.
DataProvider		See 4.8.2.2.
	hasAgreement	Association to the Provision Agreements for which the provider supplies data or metadata.
	source	Association to a data or metadata source which can process a data or metadata query.
ProvisionAgreement		Links the data provider to the relevant Structure Usage (e.g. Dataflow Definition or Metadataflow Definition) for which the provider supplies data or metadata The agreement may constrain the scope of the data or metadata that can be provided.
	source	Association to a data or metadata source which can process a data or metadata query.

Class	Feature	Description
Datasource	Abstract class: Sub classes are: *QueryDatasource* SimpleDatasource	Identification of the location or service from where data or metadata can be obtained.
	sourceURL	The URL of the data or metadata source.
QueryDatasource	Abstract class: Inherits from: *Datasource* Sub classes are: RestDatasource WebServiceDatasource	A data or metadata source which can process a data or metadata query.
RestDatasource		A data source that is accessible via a Rest interface.
WebService Datasource		A data source that conforms to a web service interface.
	wsdlURL	The URL of the web service definition language profile of the web service.
Registration		This is not detailed here but is shown as the link between the SDMX-IM and the Registry Service API. It denotes a data or metadata registration document.

10.5 Reporting Taxonomy

10.5.1 Class Diagram

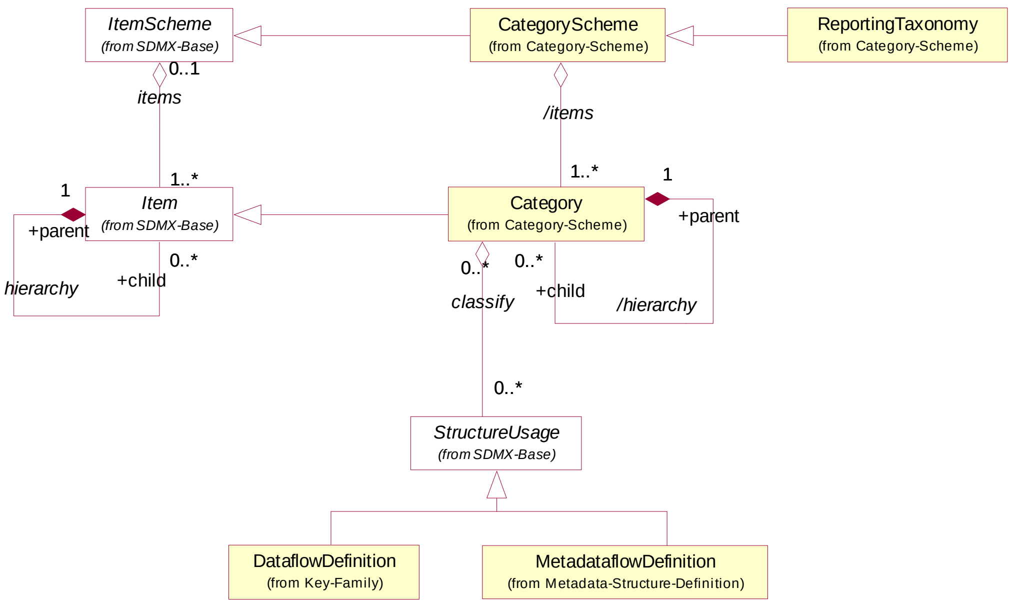


Figure 40: Class diagram of the Reporting Taxonomy

10.5.2 Explanation of the Diagram

10.5.2.1 Narrative

In some data reporting environments, and in particular those in primary reporting, the report may comprise a variety of heterogeneous data, each described by a different Structure. The definition of the set of linked sub reports is supported by the `ReportingTaxonomy`.

The `ReportingTaxonomy` is a specialised form of `CategoryScheme`. Each `Category` of the `ReportingTaxonomy` can link to a *`StructureUsage`* which itself can be one of `DataflowDefinition`, or `MetadataflowDefinition`. It is expected that within a specific `ReportingTaxonomy` each `Category` that is linked in this way will be linked to the same class (e.g. all `Category` in the scheme will link to a `DataflowDefinition`). Note that a `Category` can have child `Category` and in this way it is possible to define a hierarchical `ReportingTaxonomy`. It is possible in this taxonomy that some `Category` are defined just to give a reporting structure. For instance:

Section 1
    DatafowDefinition_1
    DatafowDefinition_2
Section 2
    DatafowDefinition_3
    DatafowDefinition_4

Here, the nodes of Section 1 and Section 2 would not be linked to `DataflowDefinition` but the other would be linked to a `DataflowDefinition` (and hence the `KeyFamily`).

10.5.2.2 Definitions

Class	Feature	Description
`ReportingTaxonomy`		A scheme which defines the composition structure of a data report where each component can be described by an independent Dataflow Definition.

11 PROCESS AND TRANSITIONS

11.1 Introduction

In any system that processes data and metadata the system itself is a series of processes and in each of these processes the data or metadata may undergo a series of transitions. This is particularly true of its path from raw data to published data and metadata. The process model presented here is a generic model that can capture key information about these stages in both a textual way and also in a more formalised way by use of expressions, possibly linked to specific identifiable objects.

11.2 Model – Inheritance View

11.2.1 Class Diagram

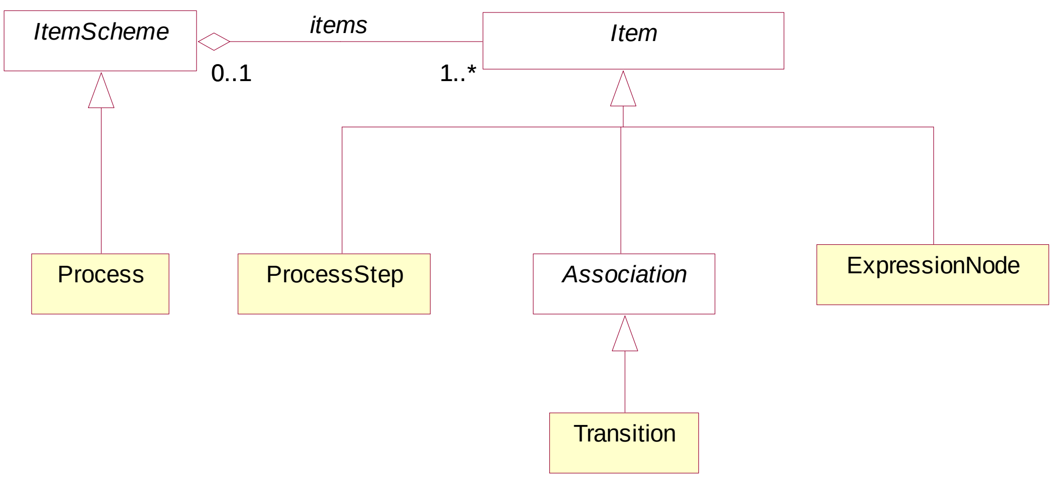


Figure 41: Inheritance class diagram of Process and Transitions

11.2.2 Explanation of the Diagram

11.2.2.1 Narrative

`Process` is an *`ItemScheme`*, `ProcessStep` is an *`Item`*, thus a `Process` is a tree of `ProcessSteps`. This implies that any `ProcessStep` can comprise an arbitrary number of sub-`ProcessSteps`. `Transition` is an *`Association`* between

ProcessSteps. ExpressionNode is also an *Item*, and is used both to describe the computations contained in the ProcessStep and to define navigation from Process to Process.

Definitions of these classes can be found below in the relationship view.

11.3 Model – Relationship view

11.3.1 Class Diagram

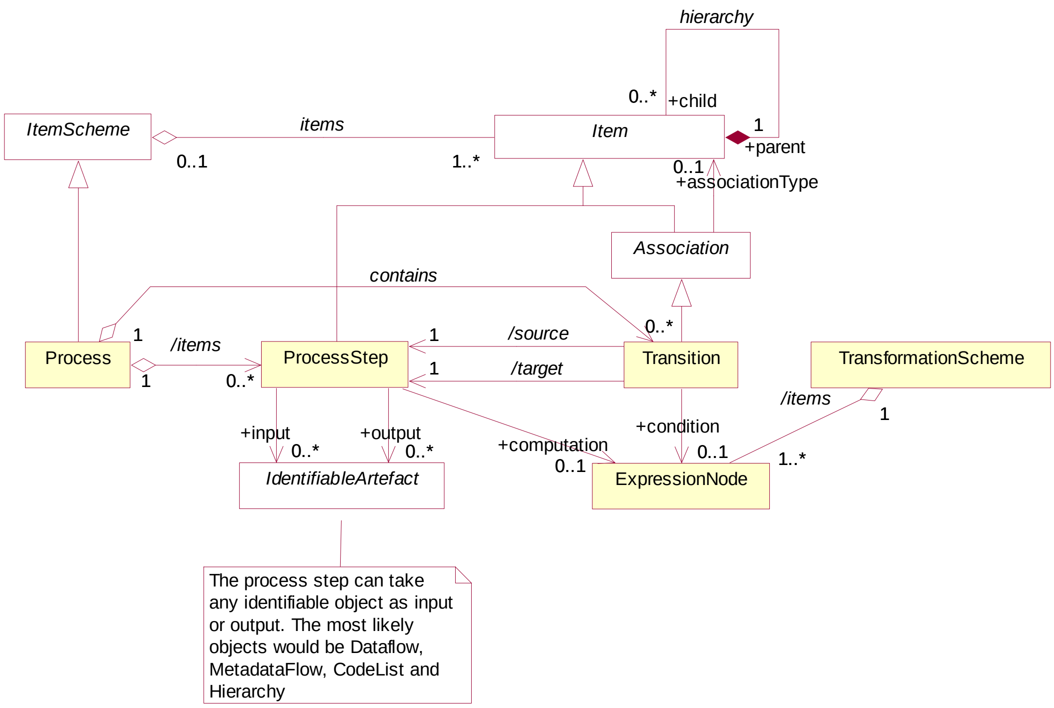


Figure 42: Relationship class diagram of Process and Transitions

11.3.2 Explanation of the Diagram

11.3.2.1 Narrative

The Process is a scheme of hierarchical ProcessSteps. Each ProcessStep can take zero or more *IdentifiableArtefacts* as input and output. Practically speaking, these are most likely to be DataflowDefinitions, Hierarchy and CodeLists - but could be anything in the model. The computation performed by a ProcessStep is optionally described by an ExpressionNode, which can represent an arbitrary expression involving any identifiable model objects. The ProcessStep could also be described textually in multiple languages. The Transition controls the execution of ProcessSteps from source ProcessStep to target ProcessStep based on the evaluation of a condition defined in ExpressionNode. The Transition can be used for looping and conditional execution of ProcessSteps.

The section on TRANSFORMATIONS AND EXPRESSIONS explains the structure of the ExpressionNode and TransformationScheme.

The operation performed on data in order to derive new information according to a given set of rules

11.3.2.2 Definitions

Class	Feature	Description
`Process`	Inherits from *`ItemScheme`*	A scheme which defines or documents the operations performed on data in order to validate data or to derive new information according to a given set of rules.
	`/items`	Associates the Process Steps.
	`contains`	Associates the Transitions.
`ProcessStep`		A specific operation, performed on data in order to validate or to derive new information according to a given set of rules.
	`+input`	Associates the Identifiable Artefacts that are inputs to the Process Step.
	`+output`	Associates the Identifiable Artefacts that are output of the Process Step.
`Transition`		An expression in a textual or formalised way of the transformation of data between two specific operations performed on the data.
	`/source`	Associates the Process Step that is the source of the Transition.
	`/target`	Associates the Process Step that is the target of the Transition.
	`+condition`	Associates an Expression Node.

12 TRANSFORMATIONS AND EXPRESSIONS

12.1 Scope

This purpose of this package in the model is to be able to track the derivation of data. It is similar in concept to lineage in data warehousing – i.e. how data is acquired or derived.

The functionality of this part of the model allows the identification and documentation of the functions performed (these will normally be automated, program functions), as well as defining structures that support a syntax neutral expression “grammar” that can specify the functions at a granular level such that a program can “read” the metadata and compose the function required in whatever computer language is appropriate.

It should be noted that the model represented above is similar in scope and content to the Expression metamodel in the Common Warehouse Metamodel (CWM) developed by the Object Management Group (OMG). This specification can be found at:

http://www.omg.org/cwm

The Expression metamodel is described in Section 8.5 of Part 1 of the CWM specification. The class diagram shown below is an interpretation of the CWM Expression metamodel expressed in the base classes of the SDMX-IM.

12.2 Model - Inheritance View

12.2.1 Class Diagram

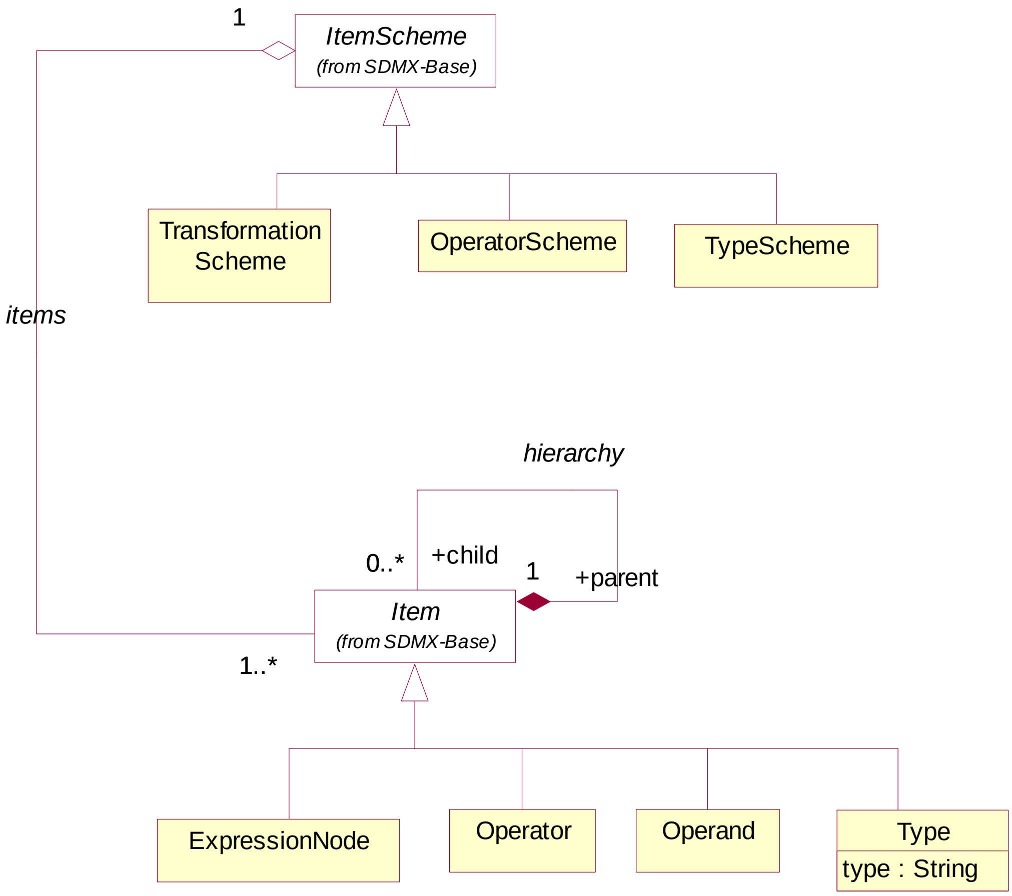


Figure 43: Inheritance class diagram of transformation classes

12.2.2 Explanation of the Diagram

12.2.2.1 Narrative

There are three type of *`ItemScheme`* relevant to this model.

1. A `TransformationScheme` which comprises one or more `ExpressionNodes`.
2. An `OperatorScheme` which comprises one or more *`Operator`*s whose `Operands` are child *`Item`*s of the `Operator`.
3. A `Type` scheme which contains, as `Types`, the expected representation of the result of the expression.

12.3 Model - Relationship View

12.3.1 Class Diagram

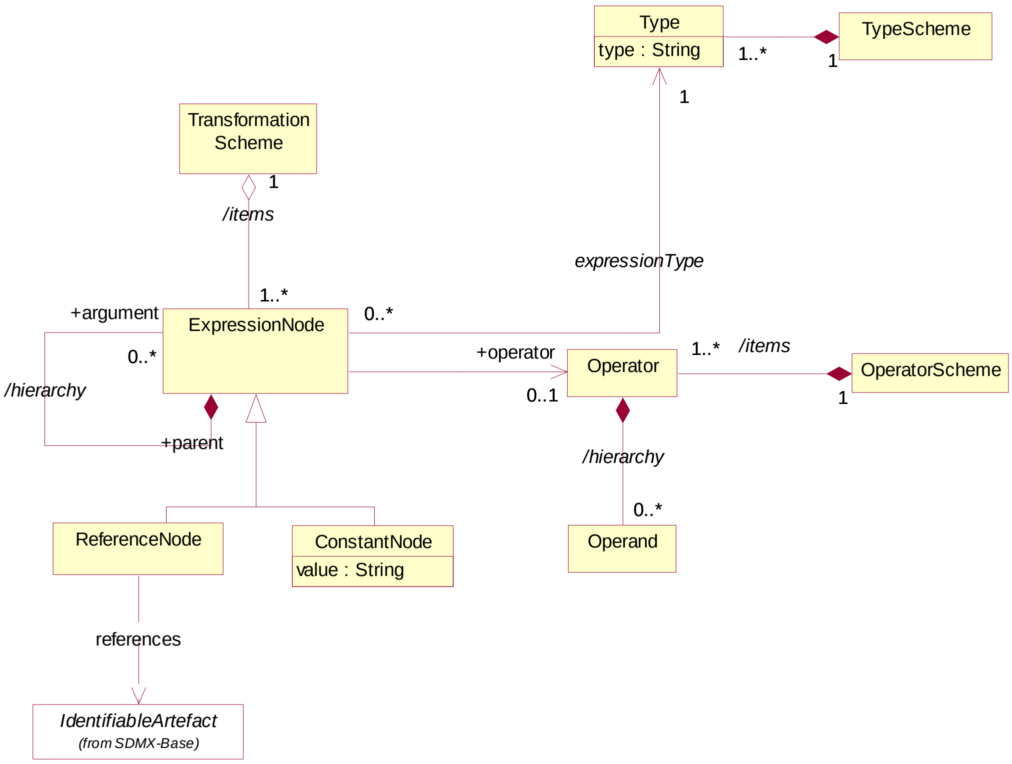


Figure 44: Relationship class diagram of expressions

12.3.2 Explanation of the Diagram

12.3.2.1 Narrative

The model presented here is a basic framework which can be used for expressions and transformations, but requires more work on elaborating its integration into the model and its actual use within the model. This elaboration will be in a future release of the standard, and may require harmonisation on contents of the `OperatorScheme` and `TypeScheme`.

The expression concept in the SDMX-IM takes a functional view of expression trees, resulting in the ability of relatively few expression types to represent a broad range of expressions. Every function or traditional mathematical operator and operand that appears in an expression hierarchy is represented by the `+operator` role on the association to `Operator` (which has as child items the `Operands`). For example, the arithmetic plus operation “a + b” can be thought of as the function “sum(a, b).” The “sum” is the `Operator`, and a and b are the `Operands`. The actual semantics of a particular function or operation are left to specific tool implementations and are not captured by the SDMX-IM.

The hierarchical nature of the SDMX-IM representation of expressions is achieved by the recursive nature of the `ExpressionNode` association. This association allows the sub-hierarchies within an expression to be treated as actual parameters of their parent nodes.

The model can be used equally to define data derivations and to define integrity checks (e.g. the Sum of A+B must equal C).

The expected format of the result of the expression (i.e. representation) is supported by the association to a `Type` defined in a scheme of types.

Although the model defines the data structures that are used to contain a syntax neutral expression, the model itself does not specify a syntax neutral expression grammar. Alternatively, the function can be described in a text form either as an unstructured explanation of the function, or as a more formal language like BNF[2]. A textual definition or description is supported because the `ExpressionNode` is a *`VersionableArtefact`* (as it inherits from *`Item`*), and thus can have multilingual descriptions.

The data structures work as follows:

The actual mathematical functions that need to be performed (e.g. sum, multiply, divide, assign (=, <, >) etc.) and their formal parameters are defined in an `OperatorScheme` which comprises one or more `Operators` each of which is a mathematical operator whose `Operands` are child Items of the `Operator` and which, together with the `Operator` define the contents of an expression.

The expressions are defined in a hierarchic `TransformationScheme` comprising `ExpressionNodes`.

The `ExpressionNode` references an `Operator` in the `OperatorScheme`. The number of child `Operands` that the `Operator` has defines the number and ordering of formal parameters that the `Operator` takes. When an `ExpressionNode` refers to an `Operator`, it must define child `ExpressionNodes` corresponding to each of the formal parameters of the `Operands` in the correct sequence. The formal parameters and corresponding arguments may be aggregate constructs such as a multi-dimensional key definition which will have the implied semantic of a `KeyDescriptor` (of `KeyFamily`).

The (child) `ExpressionNode` can have further `ExpressionNodes` defined (recursive), each of which can be a `Constant` or can be a reference to an *`IdentifiableArtefact`* (`ReferenceNode`), or another `ExpressionNode`. All *`IdentifiableArtefacts`* in the SDMX-IM have a unique urn comprising the values of the individual objects that identify it. The structure of this urn is defined in the Registry Specification. An example would be the urn of a code which comprises the agency:code-list-id.code-id – an actual example is "urn:sdmx:org.sdmx.infomodel.codelist.Code=TFFS:CL_AREA.1A").

2 BNF: Backus Naur Form

Note that it is possible using the `ReferenceNode` to identify a complete key of a measure value (by referencing a specific `KeySet` specified in a *`Constraint`* (see later)).

By way of example, the following instance diagram shows one representation of an SDMX-IM expression tree for the well-known Einstein equation E = mc2. To better understand how the equation is mapped into the expression tree, the formula can be rewritten in a functional notation as:

Assign(E, Multiply(m, Power(c, 2)))

This functional form of the equation is then mapped into a set of `ExpressionNode` instances as shown in the following figure.

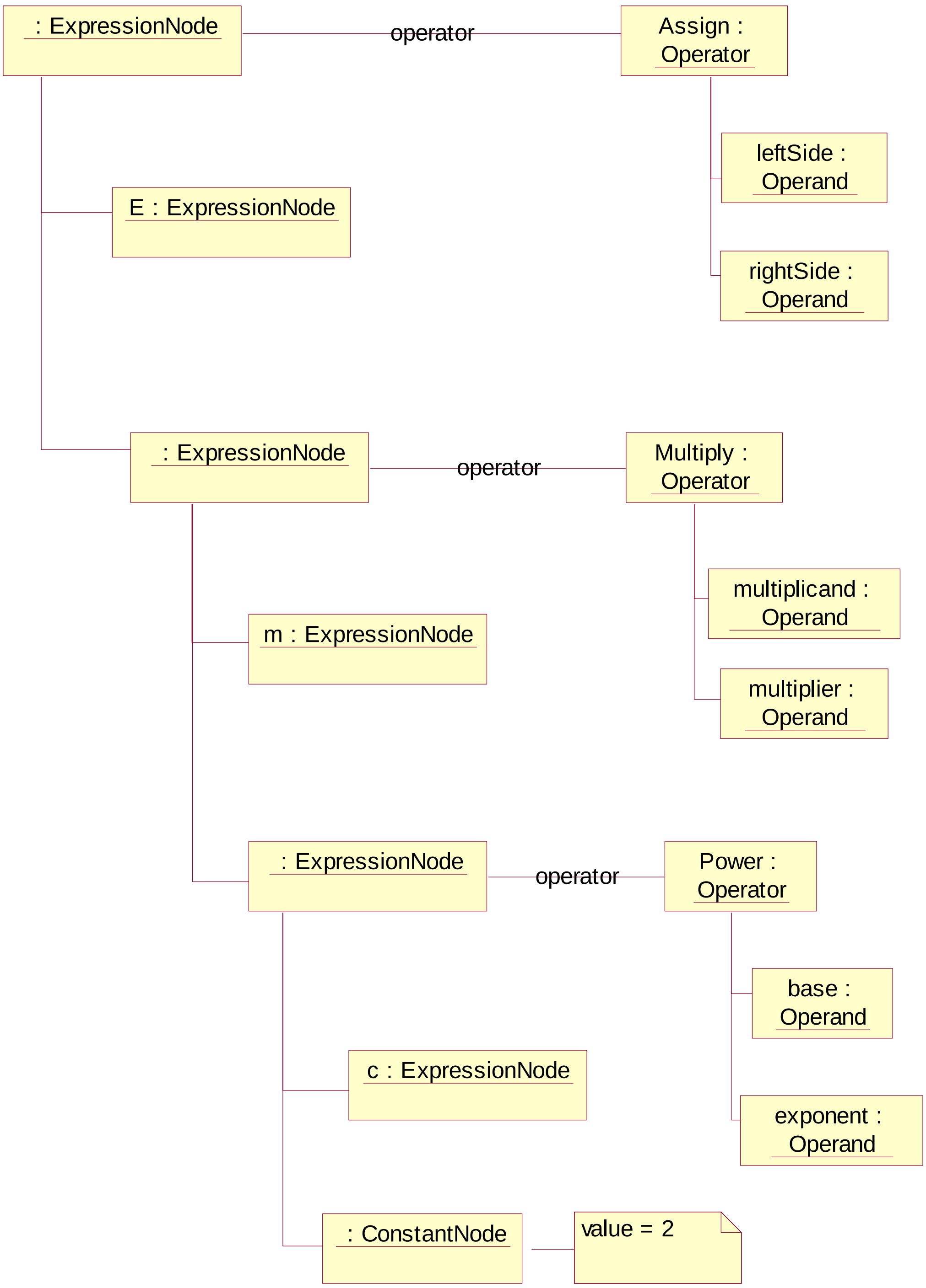


Figure 45: Collaboration diagram showing the expression $E=mc^2$

2177 #### 12.3.2.2 Definitions

Class	Feature	Description
TransformationScheme	Inherits from *ItemScheme*	A scheme which defines or documents the transformations required in order to derive or validate data from other data.
OperatorScheme	Inherits from *ItemScheme*	A scheme which defines mathematical operators and operands.
ExpressionNode	Inherits from *Item*	A node in a hierarchy of nodes that together define or document an expression.
	expressionType	Association to a Type which identifies the expected format of the result of the expression.
	+operator	Association to an Operator and its child Operands that define the mathematical operator of the Expression Node.
	+arguments	The mathematical arguments of an Expression Node.
Constant	Inherits from ExpressionNode	A specific type of Expression Node that contains a constant value.
ReferenceNode	Inherits from ExpressionNode	A specific type of Expression Node that references a specific object.
	references	Association to the Identifiable Artefact that is the referenced object.
Operator		The mathematical operator in an Operator Scheme.
	/hierarchy	Association to the Operands of the Operator.
Operand		The mathematical operand in an Operator Scheme.

2178

13 APPENDIX 1: A SHORT GUIDE TO UML IN THE SDMX INFORMATION MODEL

13.1 Scope

The scope of this document is to give a brief overview of the diagram notation used in UML. The examples used in this document have been taken from the SDMX UML model.

13.2 Use Cases

In order to develop the data models it is necessary to understand the functions that require to be supported. These are defined in a use case model. The use case model comprises actors and use cases and these are defined below.

The **actor** can be defined as follows:

> *"An actor defines a coherent set of roles that users of the system can play when interacting with it. An actor instance can be played by either an individual or an external system"*

The actor is depicted as a stick man as shown below.

Figure 46 Actor

The **use case** can be defined as follows:

> *"A use case defines a set of use-case instances, where each instance is a sequence of actions a system performs that yields an observable result of value to a particular actor"*

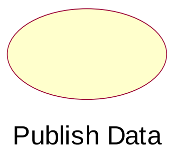


Figure 47 Use case

Figure 48 Actor and use case

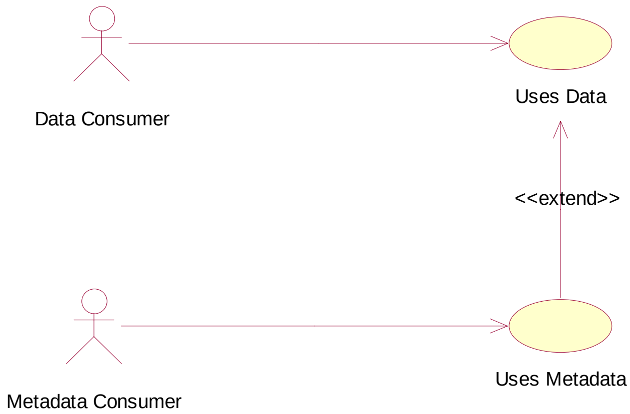


Figure 49 Extend use cases

An extend use case is where a use case may be optionally extended by a use case that is independent of the using use case. The arrow in the association points to he owning use case of the extension. In the example above the Uses Data use case is optionally extended by the Uses Metadata use case.

13.3 Classes and Attributes

13.3.1 General

A class is something of interest to the user. The equivalent name in an entity-relationship model (E-R model) is the entity and the attribute. In fact, if the UML is used purely as a means of modelling data, then there is little difference between a class and an entity.

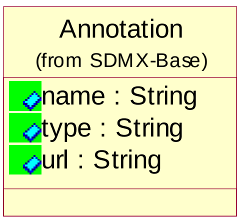


Figure 50 Class and its attributes

Figure 50 shows that a class is represented by a rectangle split into three compartments. The top compartment is for the class name, the second is for attributes and the last is for operations. Only the first compartment is mandatory. The name of the class is `Annotation`, and it belongs to the package SDMX-Base. It is common to group related artefacts (classes, use-cases, etc.) together in packages. . `Annotation` has three "String" attributes – `name`, `type`, and `url`. The full identity of the attribute includes its class e.g. the name attribute is `Annotation.name`.

Note that by convention the class names use `UpperCamelCase` – the words are concatenated and the first letter of each word is capitalized. An attribute uses `lowerCamelCase` - the first letter of the first (or only) word is not capitalized, the remaining words have capitalized first letters.

13.3.2 Abstract Class

An abstract class is drawn because it is a useful way of grouping classes, and avoids drawing a complex diagram with lots of association lines, but where it is not foreseen that the class serves any other purpose (i.e. it is always implemented as one of its sub classes). In the diagram in this document an abstract class is depicted with its name in italics, and coloured white.

Figure 51 Abstract and concrete classes

13.4 Associations

13.4.1 General

In an E-R model these are known as relationships. A UML model can give more meaning to the associations than can be given in an E-R relationship. Furthermore, the UML notation is fixed (i.e. there is no variation in the way associations are drawn). In an E-R diagram, there are many diagramming techniques, and it is the relationship in an E-R diagram that has many forms, depending on the particular E-R notation used.

13.4.2 Simple Association

Figure 52 A simple association

Here the `Dimension` class has an association with the `Concept` class. The diagram shows that a `Dimension` can have an association with only one `Concept` (1) and that a `Concept` can be linked to many `Dimensions` (0..*). The association is sometimes named to give more semantics.

In UML it is possible to specify a variety of "multiplicity" rules. The most common ones are:

- Zero or one (0..1)
- Zero or many (0..*)
- One or many (1..*)
- Many (*)
- Unspecified (blank)

13.4.3 Aggregation

Simple Aggregation

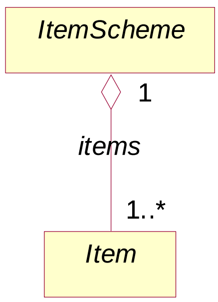


Figure 53 A simple aggregate association

An association with an aggregation relationship indicates that one class is a subordinate class (or a part) of another class. In an aggregation relationship, the child class instance can outlive its parent class. To represent an aggregation relationship, draw a solid line from the parent class to the subordinate class, and draw an unfilled diamond shape on the parent class's association end. Figure 53 shows an example of an aggregation relationship between an `ItemScheme` and an Item.

Composition aggregation

The composition aggregation relationship is just another form of the aggregation relationship, but the child class's instance lifecycle is dependent on the parent class's instance lifecycle. In Figure 54, which shows a composition relationship between a `ComponentStructure` class and a `ComponentList` class, notice that the composition relationship is drawn like the aggregation relationship, but this time the diamond shape is filled.

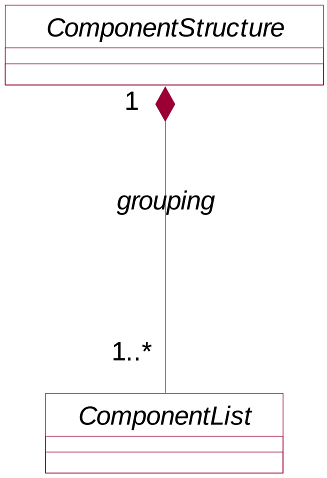


Figure 54 An aggregate association by composition

13.4.4 Association Names and Association-end (role) Names

It can be useful to name associations as this gives some more semantic meaning to the model i.e. the purpose of the association. It is possible for two classes to be joined by two (or more) associations, and in this case it is extremely useful to name the purpose of the association. Figure 55 shows a simple aggregation between `CategoryScheme` and `Category` called */items* (this means it is derived from the association between the super classes – in this case between the *`ItemScheme`* and the *`Item`*, and another between `Category` called */hierarchy*.

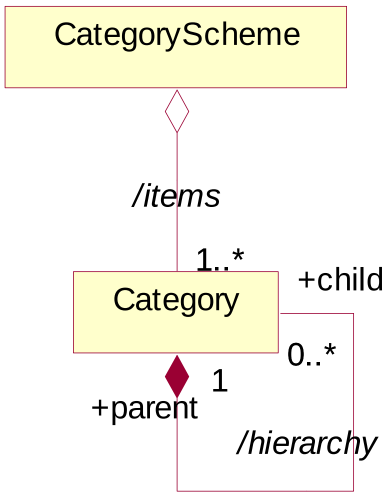


Figure 55 Association names and end names

Furthermore, it is possible to give role names to the association-ends to give more semantic meaning – such as parent and child in a tree structure association. The role is shown with “+” preceding the role name (e.g. in the diagram above the semantic of the association is that a `Category` can have zero or one parent `Category` and zero or many child `Category`).

13.4.5 Navigability

Associations are navigable in both directions. For a data model it is not necessary to give any more semantic than this. However, if there is an intent to implement the model in a database or message structure, it can be useful to identify when the association is not navigable (i.e. there is no intention or necessity to implement a navigation in a particular direction).

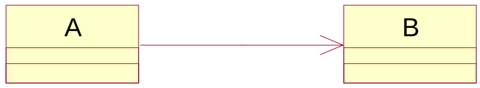


Figure 56 One way association

Here it is possible to navigate from A to B, but there is no need (e.g. no functional need) to navigate from B to A using this association.

13.4.6 Inheritance

Sometimes it is useful to group common attributes and associations together in a super class. This is useful if many classes share the same associations with other classes, and have many (but not necessarily all) attributes in common. Inheritance is shown as a triangle at the super class.

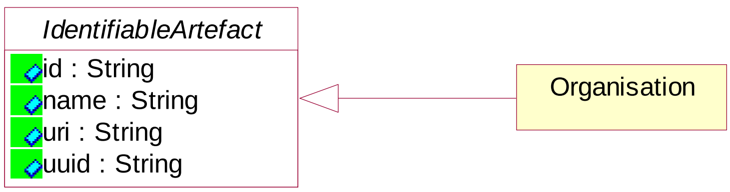


Figure 57 Inheritance

Here the Organisation is derived from *`IdentifiableArtefact`*, which is an abstract superclass. This class inherits the attributes and associations of the super class. Such a super class can be a concrete class (i.e. it actually exists), or an abstract class.

13.4.7 Derived association

It is often useful in a relationship diagram to show associations between sub classes that are derived from the associations of the super classes from which the sub classes inherit. A derived association is shown by “/” preceding the association name e.g. *`/name`*.

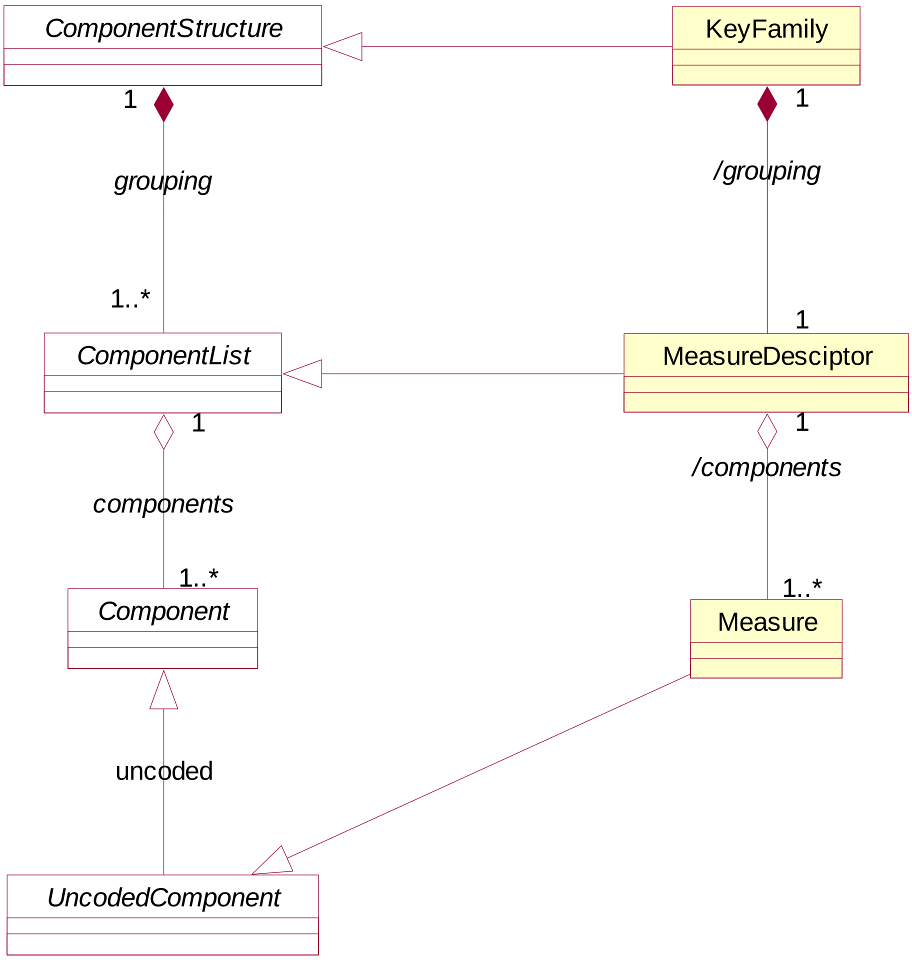


Figure 58 Derived associations

Note that the multiplicity at the association ends can be made more restrictive in the derived association. In the example above the `grouping` association is 1..* whereas the /`grouping` association is 1.

13.5 Collaboration Diagram

A collaboration diagram shows an example of an instance of the classes (an instance of a class is called an object). An instance of a class is class with a unique name.

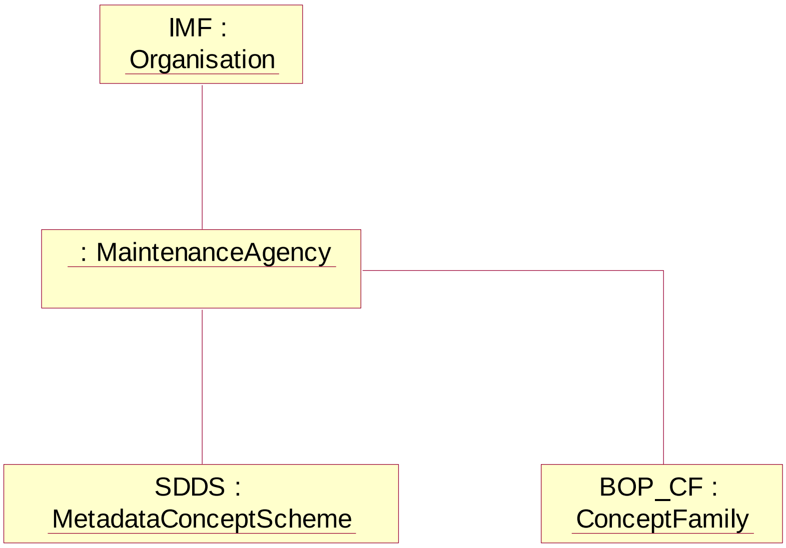


Figure 59 Collaboration diagram

Here there is an object of the `Organisation` class called IMF. In its role as `MaintenanceAgency` the IMF maintains a `MetadataConceptScheme` called SDDS and `ConceptFamily` called BOP_CF.

Sometimes it is not useful to give a name to an object. Here the object is still an instance of the class (e.g. `MaintenanceAgency`) but there is no name – so it means "any" or "it does not matter which name".

Objects are joined together using an object link.

14 APPENDIX II: KEY FAMILIES – A TUTORIAL

14.1 Introduction

This document is intended to explain "key families" to those who are completely unfamiliar with the concept. Key families are an important part of the SDMX family of standards for exchanging statistical data, and they are modelled and explained in much greater detail in other documents. However, those documents are not written to explain the basics, and will make difficult reading for those new to the idea. This document provides a basic tutorial, to help provide the basic level of understanding needed to make sense of the SDMX standards.

14.2 What is a Key Family?

In order to answer this question, we need to look at statistical data. Statistical data is represented with numbers, such as:

17369

If you are presented with a number - as above - you will have no idea of what it actually represents. You know that it is a piece of statistical data, and therefore is a measurement of some phenomenon - also known as an "observation" - but you can't tell from the number alone what it is a measurement of. A number of questions come immediately to mind:

- What is the subject of the measurement?
- What units does it measure in?
- What country or geographical region, if any, does it apply to?
- When was the measurement made?

The list of questions is potentially endless. Behind each of these questions is a particular idea, or "concept", which is used to describe the data. In our questions above, these descriptor concepts are Subject, Unit of measure, Country, and Time. If I tell you the answers to these questions, the data will begin to make sense:

- the Subject is "total population"
- the Unit of measure is "thousands of people"
- the Country is "Country ABC"
- the Time is "1 January 2001"

This is a simplified and fictional example, but it does demonstrate how we can begin to make sense of statistical data with a set of descriptor concepts. We now know that our number represents the fact that the total population of Country ABC on 1 January, 2001, was 17,369,000.

The simplest explanation of a key family is that it is a set of descriptor concepts, associated with a set of data, which allow us to understand what that data means. There is more to it, however.

14.3 Grouping Data

Numbers are often grouped together in various ways, to serve as useful packages of information. One very common approach is to have a set of observations - known as a "series", or a "time series" - made over time. This allows us to see trends in the phenomenon being measured. Thus, if I measure the total population in Country ABC on 1 January of every year, I can see whether the population is growing or declining.
A time series always has a "frequency". This is a descriptor concept which describes the intervals of time between observations. Usually, this is a regular interval, so that the frequency can be expressed as "annual" or "monthly" or "weekly". Sometimes, the intervals are irregular. Notice that a single observation does not have a frequency - only series of observations have frequencies. Frequency is an example of a descriptor concept which only applies to series of data.

There are other, higher-level groupings of data as well. A number of series are often grouped together into a "Group". Traditionally, the Group was known as a "Sibling Group", and it contained a set of Series which were identical except that they were measured with different frequencies. Thus, a given phenomenon would be measured as daily, monthly, and annually, and these Series, taken together, would be a "Sibling Group".

It is possible to have Groups which have variable values for descriptor concepts other than frequency, however: if I want to express the US daily exchange rate for all of the world's currencies over the past year, I have a different kind of group. All of the "frequency" descriptors would be the same - "daily" - but the descriptor concept which gives the "foreign currency" would be different for each series.

There is also a higher level of package known as a "Data Set". This represents a set of data that may be made up of several Groups. Typically, it is maintained and published by an agency, so that it becomes a known source of statistical data.

A basic structure is emerging: We have Observations, grouped into Series, which are grouped into Groups, which are grouped into Data Sets.

Note: It should be mentioned that there is another way of packaging Observations, which we call "cross-sectional" data. In cross-sectional data, a large number of related Observations are presented for a single point or period in time. This organization of data is very similar to Time Series data in the way a set of descriptor concepts can be associated with it. A Key Family can be used to describe both cross-sectional and time series data. For the purposes of this part of the tutorial, however, we will focus on time series data. Once we have described the Key Family for time series data, we will go back and see how cross-sectional data are structured.

What is a key family? (Answer #1)
A key family is a way of associating a set of descriptor concepts with a specific set of statistical data, as well as a technique for packaging or structuring that set of data into groups and sub-groups. This is only one way of understanding the structure and meaning of statistical data, but it provides us with a solid, generic model.

14.4 Attachment Levels

Some descriptor concepts are not meaningful at the Observation level, but only at a higher level. The example we saw earlier was frequency, which means nothing for a single Observation, but has meaning when applied to a Series of Observations. This is because it represents the interval of time between Observations. Time, on the other hand, is meaningful at the Observation level - every Observation is associated with a specific point or period in Time. Key families provide information about the level at which a particular descriptor concept is attached: at the Observation level, the Series level, the Group level, or the Data Set level. This is known as the "attachment level" of the descriptor concept.

If we think about Groups, particularly, we can see how this works. Within a group, some descriptor concepts have values that are the same for all Series within the Group, while other descriptor concepts are changeable. For the Group described above, of all US exchange rates measured daily for all of the world's currencies, the descriptor concepts of Subject ("US exchange rate") and Frequency ("daily") will be the same for all members of the Group. The descriptor concept "Foreign Currency", however, will change for each Series within the group: there will be a Series for "Swiss Francs," a Series for the "Euro," a Series for "New Zealand dollars," etc.

The rule is that descriptor concepts are “attached” to the grouping level where they become variable. Thus, if, within a single set of data, all the contents of a Series share a single value for a descriptor concept, then that descriptor concept should be attached at the Series level. This rule also assumes that the chosen level is the highest structural level where all sub-groups will share the same value. (While it is true that all Series in a Group where the country is “Switzerland” share a single value, if every Group in the Data Set would always also have the value “Switzerland” for country, then the attachment level should be the Data Set, not the Group.)

Attachment levels of descriptor concepts are always at least at the level where the concept is meaningful: thus, you cannot attach the descriptor concept frequency at the Observation level, because as a concept it only operates at the level of Series (that is, with multiple Observations made over time).

14.5 Keys

A "key family" is so called because of the term "key". "Key" refers to the values for the descriptor concepts which describe and *identify* a particular set of data. Let's take a simple example:

I have a set of statistical data which uses the following descriptor concepts:

- Time
- Frequency
- Topic
- Country

Time is always attached at the Observation level - the value for Time is the time at which the Observation was made. Time - because it is a concept connected to all statistical data - does not form part of the key. The other descriptor concepts - frequency, topic, and country - are all attached at the series level. For any given Series of Observations, they will all have a single value.

If we have a Series of data which is the monthly measurement of the total population of Country ABC, we will have a key made up of the following values for each descriptor concept:

Frequency = "monthly"
Topic = "total population"
Country = "Country ABC"

This set of values - "Monthly - total population - Country ABC" is the "key" for this data Series: it identifies what the data is.

Keys are most often associated with data at the Series level, but they also exist at other levels. For example, we could enlarge our example to be a Group including the monthly total population data for all of the countries in the world. At the Group level, Frequency would have a value of "monthly", and Topic would have a value of "total population", but we would not specify the Country descriptor concept, because it would change from Series to Series. The key for the Group is known as a "Group Key" - it identifies what the Group is, rather than identifying the Series. (In order to completely understand the Group, of course, we also need to know which descriptor concepts are changeable - in this case, Country.)

The key values are attached at the Series level, and are given in a fixed sequence. Frequency is the first descriptor concept, and the other concepts are assigned an order for that particular data set. This makes it much easier to share and understand statistical data.

If you look back to our initial use of this example, you will notice that we have not been discussing the "Unit of measure" descriptor concept. This is because the "key" only contains values for those descriptor concepts which identify the data. If we have the measurements made in thousands or in millions, the data are the same - they can be derived from one another by simply multiplying the numbers in the data by the appropriate conversion factor.

This points out a major distinction between the two types of descriptor concepts: the ones which both identify and describe the data are called "dimensions", and those which are purely descriptive are called "attributes". Only "dimensions" - that is, the descriptor concepts which also identify the data - are used in the "key", because the "key" is fundamentally a way of identifying a set of data.

14.6 Code Lists and Other Representations

In order to be able to exchange and understand data, a key family tells us what the possible values for each dimension are. This list of possible values is known as a "code list." Each value on that list is given a language-independent abbreviation - a "code" - and a language-specific description. This helps us avoid problems of translation in describing our data: the code can be translated into descriptions in any language without having to change the code associated with the data itself. Wherever possible, the values for code lists are taken from international standards, such as those provided by ISO for countries and currencies.

As stated, dimensions are always represented with codes. Attributes are sometimes represented with codes, but sometimes represented by numeric or free-text values.

This is allowed because the attributes do not serve an identification function, but merely describe the data.

What is a key family? (Answer #2)

We now have a more sophisticated understanding of a what a key family does: it specifies a set of concepts which describe and identify a set of data. It tells us which concepts are dimensions (identification and description), and which are attributes (just description), and it gives us an attachment level for each of these concepts, based on the packaging structure (Data Set, Group, Series, Observation). It also tells us which code lists provide possible values for the dimensions, and gives us the possible values for the attributes, either as code lists or as numeric or free text fields.

14.7 Cross-Sectional Data Structures

Given the explanation of Key Families thus far, we understand that a Key Family associates descriptor concepts with data, some of which also serve to identify the data – the "dimension" concepts which make up the Key.

Cross-sectional data structures do not apply a different set of concepts to the data: the same concepts still apply in describing and identifying the data. It attaches the concepts to the data differently, to create a different presentation of the data.

If we go back to our earlier example, we had the following concepts:

- Time
- Frequency
- Topic
- Country

If we want to take a set of data which is described and identified by this set of concepts, and present it in a cross-sectional fashion, we would not change these concepts – we would merely change the way in which they are represented – that is, attached – to the data structure.

Take, as an example, the total population of each country in the world on January 1, 2001 as a set of data. In our earlier example, we measured the population of Country ABC over a period of years – that is, over time. Time was the concept we used to organize our data in a sequence of observations.

If we organize our data to reflect only a single point in time – in this case, January 1, 2001 – then organizing our data over time makes less sense. It is still a possible way to structure the data, but we may wish to view it as a cross-section.

Think about the term "cross-section" – it can be understood to mean a group of parallel series over time, from which a section is taken, across time. Thus, a cross-section is created.

In our example, it is easy to see how this applies: instead of organizing our data over time – that is, using the time concept - we are choosing to organize it over the Country concept. Thus, instead of having a single value for Frequency, Topic, and Country for all Observations in our series, with a Time value associated with each Observation, we will have a Country value associated with each Observation, and a single value for Frequency, Topic, and Time. Instead of calling the group of Observations a “Series”, we now use the term “Section”.

In our earlier example, we had a key which existed mostly at the Series level:

Frequency	=	"monthly"
Topic	=	"total population"
Country	=	"Country ABC"

Time – our remaining concept, was associated with the Observations, with a different value for each one. Thus, we could have a Series which looks like this:

January 1, 2001	–	17369
February 1, 2001	–	17370
March 1, 2001	–	17405

For our cross-sectional presentation, we would have most of our key at the Section level (or, potentially, at a higher level of grouping):

Frequency	=	"monthly"
Topic	=	"total population"
Time	=	"January 1, 2001"

With each Observation, we now have a Country value, instead of a Time value:

Country ABC	=	“17369”
Country XYZ	=	“24982”
Country HIJ	=	“37260”

In this cross-sectional presentation of our data set, we have chosen to present each Observation paired with a Country value, taken from our Codelist of values for the concept Country. Other dimensions could as easily produce a cross-sectional view, by attaching their values at The Observation level, instead of the values for Country, as in our example.

Because the concepts themselves do not change, but only the way in which they are attached to the data structure, a single key family can be used to describe both time-series and cross-sectional presentations.

In the version 1.0 SDMX standards, formats are capable of presenting cross-sectional data for any single dimension concept, as well as presenting the data as a time series. It is up to the key family creator to select which non-Time concept, used as a dimension, will serve to organize a cross-sectional presentation. In future versions, it is possible that more complete support for the possible cross-sectional views for a key family will be provided.