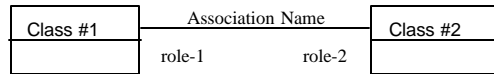


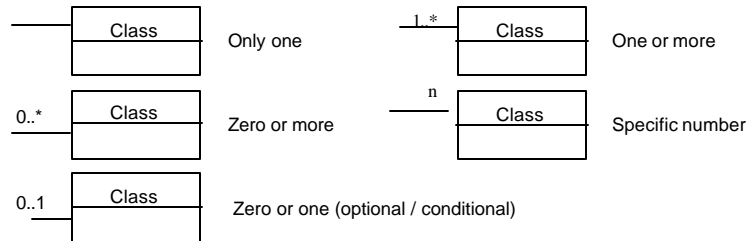
1.1 UML notations

The diagrams that appear in this standard are presented using the Unified Modelling Language (UML) static structure diagram and the UML Object Constraint Language (OCL) as the conceptual schema language. The UML notations used in this standard are described in the diagram below.

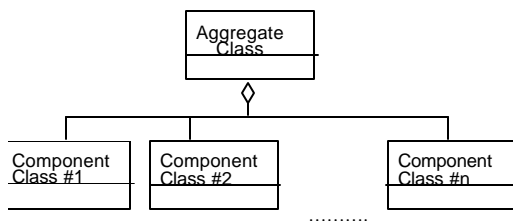
Association between classes



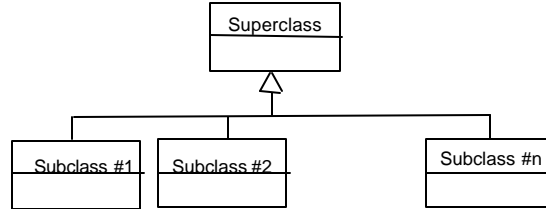
Association Multiplicity



Aggregation between classes



Class Inheritance (subtyping of classes)



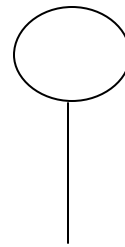
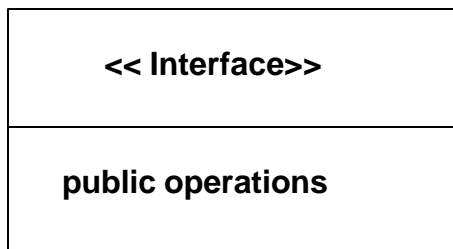
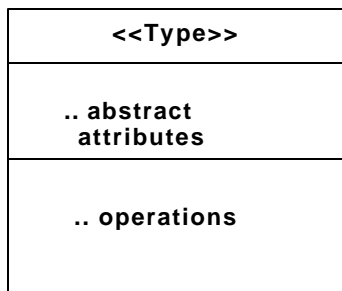
1.2 Classes

A **class** is a descriptor of a set of objects that share the same attributes, operations, methods, relationships, and behavior. A class represents a concept within the system being modeled. Depending on the kind of model, the concept may be based on the real world (for an analysis model), or it may also contain algorithmic and computer implementation concepts (for a design model). A classifier is a generalization of a class that includes other class-like elements, such as data types, actors, and components. A UML class has a name, a set of attributes, a set of operations, and constraints. A class may participate in associations.

A class in the ISO 19100 series is viewed as a specification and not as an implementation. Attributes are considered abstract and do not have to be directly implemented (i.e. as fields in a record or instance variables in an object).

All classes must have unique names. All Classes should be defined within a package. Class names should start with an upper case letter. A class should not have a name that is based on its external usage, since this may limit reuse. A class name should not contain spaces. Separate words in a class name should be concatenated. Each subword in a name should begin with a capital letter, such as "XnnnYmmm". Class names should be preceded by a capitalized two-letter prefix that denotes the package that contains that class, such as "XY_XnYm". A list of these prefixes can be found in sub-clause 1.8, along with the relevant ISO 19100 series standard. Basic data types (CharacterString, Number, Record...etc) need not being with a prefix.

Multiple inheritance should be used at a minimum, because it tends to increase model complexity.



**Interface
icon
“lollipop”**

Type and interface symbols

UML defines two special kinds of classes through the stereotypes <<Type>> and <<Interface>>. An Interface in UML is a named set of operations that characterizes the behavior of an element. An example of a UML Interface is the lollipop icon, shown in the figure above. A <<Type>> is a class with operation definitions that are similar to an <<Interface>>. However in a <<Type>>, it is also possible to specify abstract attributes.

An <<Interface>> in UML contains no attributes and does not imply any implementation. It is not a class in the common sense. An alternative to <<Interface>> is Abstract. An abstract class defines a polymorphic object class and cannot be instantiated. An abstract class differs from an <<Interface>> in that it may have attributes and may imply some of its simpler operation implementations. Note that an Abstract class is specified by having the classname in italics, or by the tagged value {Abstract} placed next to the classname. Since the ISO 19100 series does not focus on implementation specifications, there should be no need to use abstract classes.

Implementation specifications shall use the “realize” relation between abstract specification interfaces and implementation specification classes. Implementation specification may use the “generalize” relation between abstract specification interfaces and implementation specification interfaces. The ISO 19100 series allows for the common semantic variation of the <<interface>> stereotype in UML in which an interface is allowed to have attributes, thus making it similar to the stereotype <<Type>>. This is to be consistent with several other standards such as OMG IDL, JAVA, the ISO 19100 series and SQL/MM. It also makes the interface simpler to understand and smaller graphically. A mapping can be made back to standard UML by replacing each attribute “attributeName” with pairs of operation interfaces “getAttributeName” and “setAttributeName”.

The use of “interface realize” instead of “subclass generalization” shows that a class is implementing an interface. Since interfaces do not actually define any class information, a class cannot really inherit from an interface; it can only “implement” it. This is especially true for the interface attributes. The semantic variation described above implies that an object can either support the attributeName as an attribute member or support the get-set pair of interfaces. A class with an “interface realize” has more control over implementation inheritance than one that uses “interface generalize.” In the former, the class can inherit from any of its ancestors a semantically viable operation that is appropriately polymorphic with respect to each of the interface’s signature. In the latter, the class would have to implement to keep the name space valid.

For objects that may be passed “by value” such as in the transfer of features, it is necessary that sufficient information be provided about the abstract state. This means that an “interface only” based specification cannot be used. However, the notion of abstract attributes in an interface/type can be used as the place to describe the necessary abstract state.

1.3 Attributes

UML notation for an attribute has the form:

<<stereotype>> [visibility] name [multiplicity] [:type] [= initial value] [{property-string}]

+ center : Point = (0,0) {frozen}

+ origin [0..1] : Point // multiplicity 0..1 means that this is optional

The UML standard defined properties that can be used re: changeable, addOnly, frozen (const).

An attribute must be unique within the context of a class and its supertypes, or else be a derived attribute.

If an attribute is derived, it shall be treated as a model element that can be computed from another element, but that is shown for clarity or that is included for design purposes even though it adds no semantic information.

Attribute names should start with a lower-case letter.

All attributes must be typed and the type must exist among the set of legal base types, the constructed/defined types. A type must always be specified, there is no default type.

If no explicit multiplicity is given, it is assumed to be 1.

An attribute may define a default value, which is used when an object of that type is created. Default values are defined in the UML definition of the attribute.

UML allows definition of both attribute and operation interfaces to objects. An attribute should not be interpreted as a data or instance variable that needs to be stored physically in an implementation.

Attributes may be implemented either directly as data or as a pair of “accessor” and “mutator” operations for getting and setting values.

1.4 Data Types

1.4.1 Basic Data Types

The fundamental building blocks out of which all forms of data are composed are primitive data elements consisting of numbers, text strings and dates. A data type defines the legal value domain and the operations on values of that domain. The focus from a specification point of view is to define the intention and semantics of the types – they might be differently encoded and represented in various implementation environments, as long as the representation is able to represent the intended type. These data types are thus for the purpose of creating abstract specifications. For implementation specifications a mapping will be defined from these data types to the data types of the target environment. The following table gives a summary of the basic data types.

Summary of basic data types

Data type	Description
Integer	An integer number.
Float	A floating point real number.
Double	Double describes a signed, approximate, numeric value with a binary precision 53 (zero or absolute value 10^{-308} to 10^{308}).
Binary	A sequence of octets.
String	A string of characters.
Date	A string which follows the ISO 8601 formats for time.
Time	A string which follows the ISO 8601 formats for date.
DateTime	A string which follows the combined date / time format of ISO 8601.
Boolean	A quantity that takes the values TRUE or FALSE.
Vector	A set of number representing a coordinate in a coordinate system.

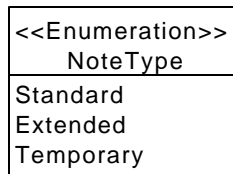
1.4.2 Enumeration Types

enum { value1, value2, value3 }

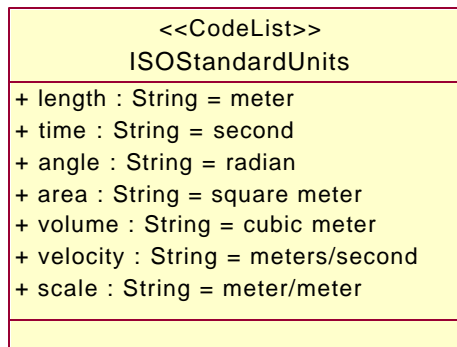
An enumerated type declaration defines a list of valid identifiers. Attributes of an enumerated type can only take values from this list.

EXAMPLE attr1 : BuildingType, where BuildingType is defined as Enum BuildingType { Public, Private, Tourist };

Enumerations are modelled as classes that are stereotyped as Enumerations. An enumeration class can only contain simple attributes which represent the enumeration values. Other information within an enumeration class is void. An enumeration is a user-definable data type, whose instances form a list of named literal values. Usually, both the enumeration name and its literal values are declared.







CodeList can be used to describe a more open enumeration. <<CodeList>> is a flexible enumeration that uses string values through a binding of the Dictionary type key and return values as string types; e.g. Dictionary (String, String). Code lists are useful to for expressing a long list of potential values. If the elements of the list are completely known, an enumeration should be used; if the only likely values of the elements are known, a code list should be used. Enumerated code lists may be encoded according to a standard, such as the ISO standard for two-letter country codes or for two-letter language code. Code lists are more likely to have their values exposed to the user, and are therefore often mnemonic. Different implementations are likely to use different encoding schemes (with translation tables back to other encoding schemes available).



Example of a CodeList

1.5 Relationships and Associations

-  **Association**
A semantic connection between two instances
-  **Generalization**
A relationship between an element and the subelements that may be substituted for it
-  **Dependency**
The use of one element by another
-  **Refinement**
A shift in levels of abstraction

A relationship in UML is a semantic connection among model elements. Kinds of relationships include association, generalization, metarelationship, flow, and several kinds grouped under dependency. To reify is to treat as an object something that is not usually regarded as an object.

In the above figure it is made a clear distinction between the general term “relationship,” and the more specific term “association”. Both are defined for class to class linkages, but association is reserved for those relationships that are in reality instance to instance linkages. “Generalization,” “realization” and “dependency” are class to class relationships. “Aggregation,” and other object to object relationships, are more restrictively called “associations.” It is always appropriate to use the most restrictive term in any case, so in speaking of instantiable relationships, use the term “association.”

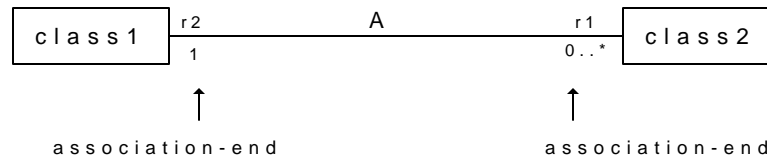
In the ISO 19100 series generalization, dependency and refinement are used according to the standard UML notation and usage. In the following the usage of association, aggregation and composition is described further.

1.5.1 Association, composition and aggregation

An association in UML is the semantic relationship between two or more classifiers (i.e. class, interface, ..) that involves connections among their instances.

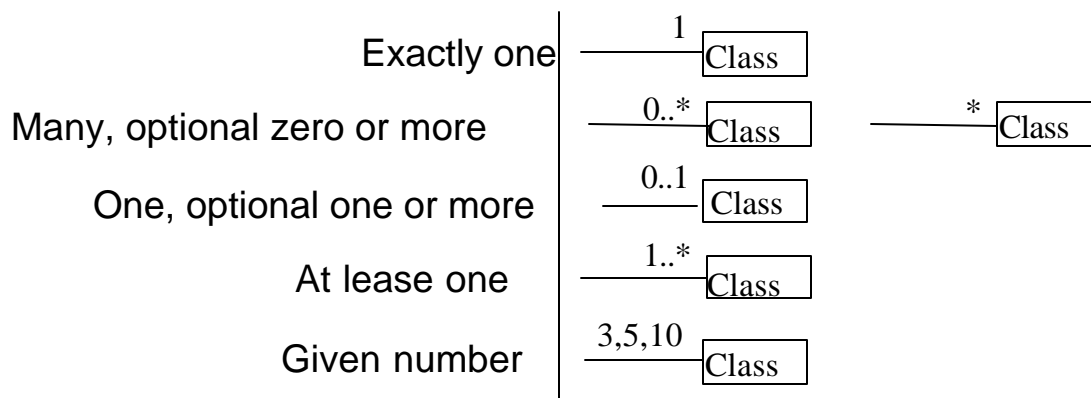
An association is used to describe a relationship between two or more classes. UML defines three different types of associations called association, aggregation and composition. The three types have different semantics. An ordinary association shall be used to represent a general relationship between two classes. The aggregation and composition associations shall be used to create part-whole relationships between two classes.

A binary association has a name and two association-ends. An association-end has a role name, a multiplicity statement, an optional aggregation symbol. An association-end shall always be connected to a class.



Association

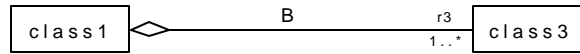
This figure shows an association named "A" with its two respective association-ends. The role name is used to identify the end of an association, the role name **r1** identifies the association-end which is connected to the class named **class2**. The multiplicity of the association-end can be one of exactly-one (1), zero-or-one (0..1), one-or-more (1..*), zero-or-more (0..*) or an interval (n..m). Viewed from the class, the role name of the opposite association-end identifies the role of the target class. We say that **class2** has an association to **class1** that is identified by the role **r2** and which as a multiplicity of exactly one. The other way around, we can say that **class1** has an association to **class2** that is identified by the role name **r1** with multiplicity of zero-or-more. In the instance model we say that **class1** objects have a reference to zero-or-more **class2** objects and that **class2** objects have a reference to exactly one **class1** object.



Specification of multiplicity/multiplicity

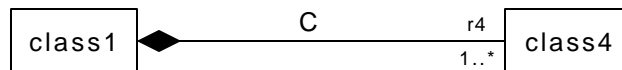
The number of instances that can participate at one end in an association (or attribute) is specified in the above Figure.

An aggregation association is a relationship between two classes, in which one of the classes plays the role of container and the other plays the role of a containee. The above figure shows an example of an aggregation. The diamond-shaped aggregation symbol at the association-end close to **class1** indicates that **class1** is an aggregation consisting of **class3**. We say that **class3** is a part of **class1**. In the instance model, **class1** objects will contain one-or-more **class3** objects. The aggregation association shall be used when the containee objects (that represent the parts of a container object) can exist without the container object. Aggregation is a symbolic short-form for the part-of association but does not have explicit semantics. It allows for sharing of the same objects in multiple aggregations. If a stronger aggregation semantics is required, composition should be used as described below.



Aggregation

A composition association is a strong aggregation. In a composition association, if a container object is deleted then all of its containee objects are deleted as well. The composition association shall be used when the objects representing the parts of a container object, cannot exist without the container object. The above figure shows a composition association in which the diamond-shaped composition symbol is has a solid fill. Here **class1** objects consist of one-or-more **class4** objects, and the **class4** objects cannot exist unless the **class1** object also exists. The required (implied) multiplicity for the owner class is always one. The containees, or parts, cannot be shared among multiple owners. Composition should be used to have the semantic effect of containment.



Composition (strong aggregation)

All associations should have cardinalities defined for both association ends. At least one role name shall be defined. If only one role name is defined, the other will by default be inv_rolename.

All association ends (roles) representing the direction of a relationship must be named or else the association itself must be named. The name of an association end (the rolename) must be unique within the context of a class and its supertypes.

The direction of an association must be specified. If the direction is not specified, it is assumed to be a two-way association. If one-way associations are intended, the direction of the association can be marked by an arrow at the end of the line.

If only the association is named, the direction of the association should be specified. The name of the association must be unique within the context of a class and its supertypes or else it must be derived.

Every UML association has navigability attributes that indicate which player in the association has direct access to the association opposite role. The default logic for an unmarked association is that it is two-way. In the case of client-server relations, the association used is normally only navigable from client to server (one-way), indicated in the diagram by an arrowhead on the server side of the association. Associations that do not indicate navigability are two-way in that both participants have equal access to the opposite role. Two-way navigation is not common or necessary in many client-to-server operations. The counterexample to this may be notification services, where the server often instigates communication on a prescribed event.

The use of two-way relations that introduce unreasonable package dependencies should be minimized. One-way relations should be used when that is all that is needed.

If an association is navigable in a particular direction, the model shall supply a “role name” that is appropriate for the role of the target object in relation to the source object. Thus in a 2-way association, two role names will be supplied. The default role name is “the<target class name>” in which the target class is referenced from the source class (this is the default name in many UML tools). Association names are of secondary importance and actually are more for documentation purposes. Sometimes they can, however, be used for generating association-manager objects in environments that support associations as a first-class citizen concept.

Multiplicity refers to the number of relationships of a particular kind that an object can be involved in. If the target class of a one-way relation is marked with a multiplicity, then this indicates to any implementation how many storage locations may be required by the source class of the relation. In most object languages, this would be implemented as an array of object references. The multiplicity indicates the constraints on the length of that array and the number of NULLs allowed. If an association end were not navigable, putting a multiplicity constraint on it would require an implementation to track of the use of association by other objects (or to be able to acquire the multiplicity through query). If this is important to the model, the association should be two-way navigable to make enforcement of the constraint more tenable. In other words, a one-way relation implies a certain “don’t care” attitude towards the non-navigable end.

A derived association must be marked as derived in its specification. A derived association and the corresponding association end(s) shall be handled according the rules of the derivation and considered as an abstract association/association end that can be computed on demand. There should be no double-specifications in which the same element is specified as both an attribute and an association. N-ary relationships, for $N > 2$ shall be avoided in order to reduce complexity. Multiplicity for associations are specified as UML multiplicity specifications.

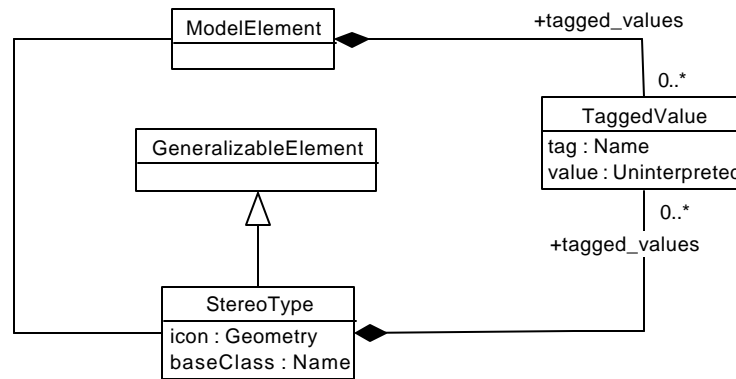
An association with role names can be viewed as similar to defining attributes for the two classes involved, with the additional constraint that updates and deletions are consistently handled for both sides. For one-way associations, it thus becomes equivalent to an attribute definition. The recommendation for the ISO 19100 series is to use the association notation for all cases except for those involving attributes of basic data types.

1.6 Stereotypes and tagged values

UML contains three extensibility mechanisms that are used to specialize the use of UML for the modelling of geographic information and services.

A *UML stereotype* is an extension mechanism for existing UML concepts. It is an model element that is used to classify (or mark) other UML elements so that they in some respect behave as if they were instances of new virtual or pseudo metamodel classes whose form is based on existing base metamodel classes. Stereotypes augment the classification mechanisms on the basis of the built-in UML metamodel class hierarchy. Therefore, names of new stereotypes must not clash with predefined metamodel elements or other stereotypes.

Tagged values are another extensibility mechanism of UML. A tagged value is a tag-value pair that can be used to add properties to any model element in UML, i.e. it can extend an arbitrary existing element in the UML metamodel or extend a stereotype.



UML stereotypes and tagged values

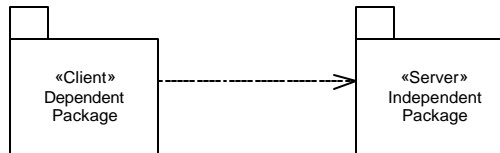
In this standard the following stereotypes are used:

- a) <<Interface>> a definition of a set of operations that is supported by objects having this interface.
- b) <<Type>> a stereotyped class used for specification of a domain of instances (objects), together with the operations applicable to the objects. A type may have attributes and associations.
- c) <<Entity>> meaning a class that represent information-carrying, potentially persistent objects. A standard UML extension from the Unified Software Development Process.
- d) <<Enumeration>> A data type whose instances form a list of named literal values. Both the enumeration name and its literal values are declared. Enumeration means a short list of well-understood potential values within a class.
- e) <<MetaClass>> A class whose instances are classes. Metaclasses are typically used in the construction of metamodels.
- f) <<DataType>> A descriptor of a set of values that lack identity (independent existence and the possibility of side effects). Data types include primitive predefined types and user-definable types. A DataType is thus a class with few or no operations whose primary purpose is to hold the abstract state of another class.
- g) <<CodeList>> can be used to describe a more open enumeration. <<CodeList>> is a flexible enumeration. Code lists are useful for expressing a long list of potential values. If the elements of the list are completely known, an enumeration should be used; if the only likely values of the elements are known, a code list should be used.
- h) <<Union>> Union describes a selection of one of the specified types. This is useful to specify a set of alternative classes/types that can be used, without the need to create a common supertype/class.
- i) <<Feature>> - a special kind of <<Entity>> which represents an abstraction of a real world phenomena (as opposed to any kind of information-carrying object) – used for the development of application schemas (see ISO 19109).
- j) <<Abstract>> - is a class (or other classifier) that cannot be directly instantiated. UML notation for this is to show the name in italics.
- k) <<Package>> - a cluster of logically related components, containing sub-packages
- l) <<Leaf>> - A package that contains definitions, without any sub-packages.

1.7 Packages

A package is a set of related types and interfaces that forms a consistent component of a software system design. Packages do not usually form a complete system since they often invoke the services provided by other packages in the system. One package, acting as a client, may use another package, acting as a server, to supply needed services. In this case, the client package is said to be *dependent* on the server package. This is indicated in the following package diagrams.

Logical View



Because of this client-server relation, inter-package dependencies define the criterion for viable application schemas. An application schema that contains any package defined in this standard shall also contain all of its dependencies.

1.8 Package abbreviations

Two letter abbreviations are used to denote the package that contains a class. Those abbreviations precede class names, connected by a “_”. The standard that those classes are located in is indicated in parentheses. A list of those abbreviations follows.

CC – Changing Coordinates (ISO 19111 – Spatial referencing by geographic identifiers)

CI – Citation (ISO 19115 - Metadata)

DQ – Data quality (ISO 19113 – Quality principles)

DS – Dataset (ISO 19115 – Metadata)

EX – Extent (ISO 19115 – Metadata)

FC – Feature Catalogue (ISO 19115 Metadata)

GF – General Feature (ISO 19109- Rules for application schema)

GM – Geometry (ISO 19107 – Spatial schema)

LI – Lineage (ISO 19113 – Quality principles)

MD – Metadata (ISO 19115 – Metadata)

PF – Feature Portrayal (ISO 19117 – Portrayal)

PS – Positioning Services (ISO 19116 – Positioning services)

SC – Spatial Reference by Coordinates (ISO 19111 – Spatial referencing by coordinates)

SI – Spatial Identification (ISO 19112 – Spatial referencing by geographic identifiers)

TM – Temporal (ISO 19108 – Temporal Schema)

TP – Topology (ISO 19107 – Spatial Schema)