HOPEX UML - XML Schemas

HOPEX V2

Information in this document is subject to change and does not represent a commitment on the part of MEGA International.

No part of this document is to be reproduced, transmitted, stored in a retrieval system, or translated into any language in any form by any means, without the prior written permission of MEGA International.

© MEGA International, Paris, 1996 - 2016

All rights reserved.

HOPEX is a registered trademark of MEGA International.

Windows is a registered trademark of Microsoft Corporation.

The other trademarks mentioned in this document belong to their respective owners.

HOPEX UML

User Guide

HOPEX V2

Information in this document is subject to change and does not represent a commitment on the part of MEGA International.

No part of this document is to be reproduced, transmitted, stored in a retrieval system, or translated into any language in any form by any means, without the prior written permission of MEGA International.

© MEGA International, Paris, 1996 - 2016

All rights reserved.

HOPEX UML and HOPEX are registered trademarks of MEGA International.

Windows is a registered trademark of Microsoft Corporation.

The other trademarks mentioned in this document belong to their respective owners.

About HOPEX UML

UML (Unified Modeling Language) is established as standard for the graphic modeling of information systems. **HOPEX UML** offers an ensemble of views and tools allowing you to model your IS conforming to version 2.3 of this standard.

The aim of this guide is to introduce you to the main functionalities of HOPEX UML.

- ✓ "Conventions Used in the Guide", page 4
- ✓ "Overview", page 5
- ✓ "Organization of UML Diagrams", page 7
- ✓ "Starting HOPEX UML", page 9

Conventions Used in the Guide

☛ *Remark on the preceding points.*

📖 *Definition of terms used.*

☺ *A tip that may simplify things.*

Compatibility with previous versions.

Things you must not do.

Very important remark to avoid errors during an operation.

Commands are presented as seen here: **File > Open**.

Names of products and technical modules are presented in bold as seen here: **HOPEX**.

OVERVIEW

HOPEX UML can be used for:

Analyzing use cases

Before designing a system, there must be a careful analysis of the functions expected of it. The system components will be used by the "actors" in the organization to perform their tasks. The various "use cases" for the system will be presented in ***use case diagrams***.

These are used as the starting point for discovering objects.

They then allow validation of the use of these objects in the interaction diagrams.

Then they provide criteria for grouping the discovered objects into "packages".

Identifying objects

Objects with a similar structure, the same behavior, and the same types of relations with other objects, are placed in the same class.

The ***class diagram*** identifies the objects involved within the system and defines their structure in terms of attributes and operations, as well as the relationships between them. The ***object diagram*** shows the instances compatible with a particular class diagram, and can be used as an example to verify it.

Describing behaviors

The ***state machine diagram*** enables definition of the behavior of an object in response to internal or external requests it may receive. It indicates each possible object state, and the reaction of the object to a given event when in that state.

The ***activity diagram*** also describes a behavior, but in terms of actions.

Representing interactions between objects

The resulting dialog that is initiated between the different objects concerned by the event can be represented in ***interaction diagrams***.

Interaction diagrams emphasize the exchanges that take place between objects.

The ***sequence diagram*** shows the same exchanges, but indicates the chronology.

The ***communication diagram*** highlights structural organization of objects that send and receive messages.

The ***interaction overview diagram*** provides a general view of control flow.

Dividing classes between packages

Once the objects are identified, it is easy to divide the classes that implement them into different packages. These classes are grouped in the ***package diagram*** so as to minimize exchanges between different packages. They meet two criteria: the first is more technical and concerns the execution environment, while the other is more structural and is related to the use it will be put to by the users for each use case.

Defining interfaces

To respect the principle of encapsulation, there is strict distribution of elements between components. This means interfaces must be provided between elements that have relationships but belong to different components.

The ***component diagram*** and the ***composite structure diagram*** present the interdependence between components or component elements.

Defining object interfaces while complying with a standard exchange protocol (CORBA2, DCOM/OLE) is key to interoperability, enabling objects developed and used in heterogeneous environments to work together.

Specifying deployment

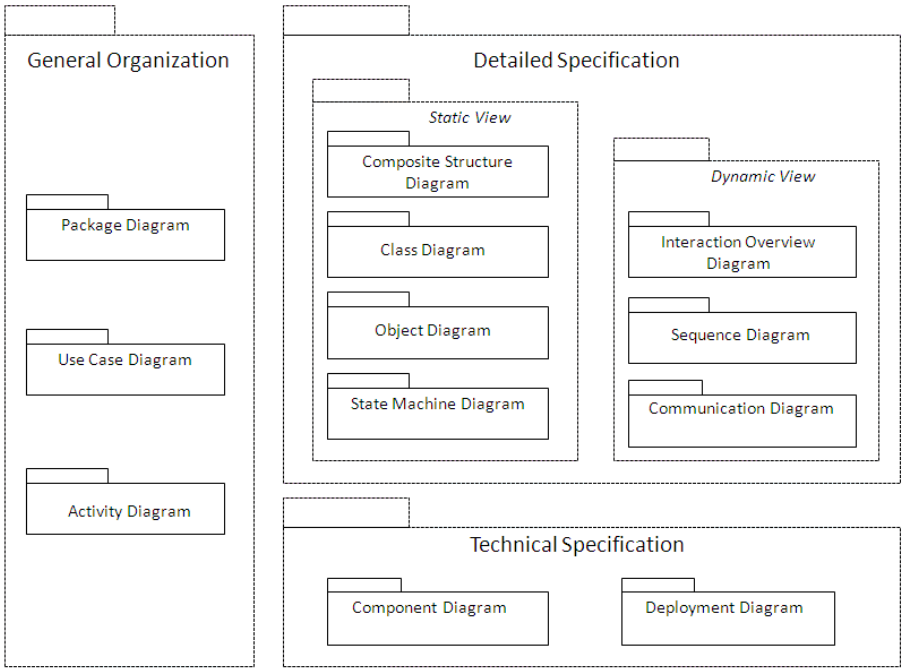
Deployment of objects in an actual work environment can be specified in the ***deployment diagram***.

Organization of UML Diagrams

General organization

Use case diagrams show the main interactions between the system being analyzed and its environment, and indicate its main sub-systems.

Package diagrams provide a more technical breakdown of the system. Dividing a system into packages imposes some structure, as an object can only be in one package. You can begin drawing package diagrams as soon as you have identified the main components of your system (Sales, Production, Invoicing, etc.).



Detailed specification

The main diagram is the ***class diagram***. It describes the essential semantics of the objects in the system. This is where designers will generally spend most of their time. Classes are generally discovered by iteration between class and sequence diagrams.

The ***state machine diagram*** describes the static aspects of an object: the different states it can be in and the possible state transitions. This fleshes out the class description.

The ***interaction diagrams*** describe the dynamic aspects of the system, by showing the interactions between objects. They provide a detailed description of the different scenarios in a use case. The sequence diagram specifies how a scenario progresses

over time, while the communication diagram stresses the interactions between objects.

Technical specification and deployment

The ***component diagram*** describes the different technical components of an application and shows their interactions.

The ***composite structure diagram*** specifies collaborations between components or component elements in execution of a common task.

The ***deployment diagram*** is used to specify the system architecture, indicating the workstations or nodes in the information system where the different application components are to be installed.

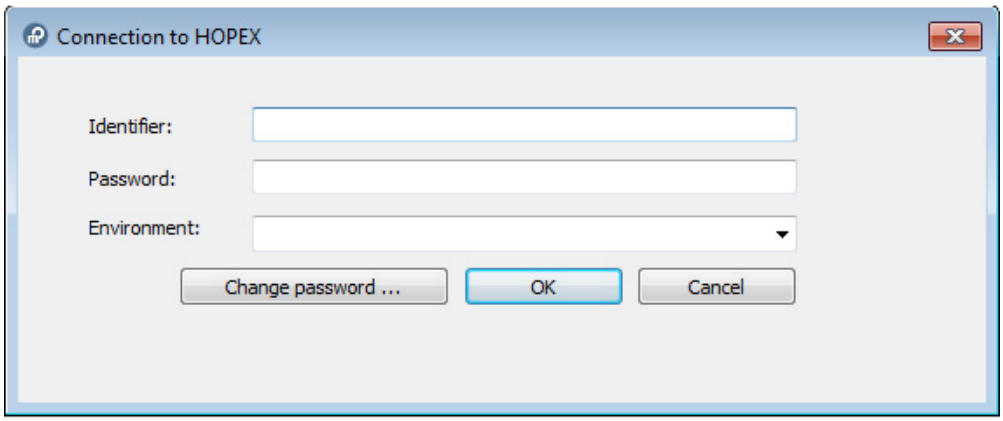
UML diagram entry points

Diagram	Entry points
Class diagram	Package, class, use case
Object diagram	Class, component, package, use case
Component diagram	Component, package
Composite structure diagram	Component, class, collaboration
Deployment diagram	Package
Package diagram	Package, library
Use case diagram	Package, use case
Sequence diagram	interaction
Communication Diagram	interaction
Interaction overview diagram	interaction
Activity diagram (UML2)	Activity
State machine diagram	State machine, , protocol state machine

STARTING HOPEX UML

Pour lancer **HOPEX** :

1. Double-cliquez sur l'icône **HOPEX** mega.exe .
   La fenêtre d'identification apparaît.

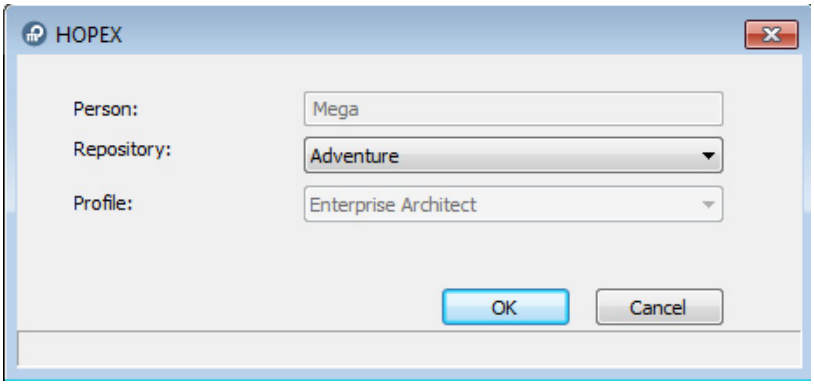


2. Dans le champ **Environnement**, sélectionnez votre environnement de travail.

   ☛ *Si vous n'avez accès qu'à un environnement, celui–ci est automatiquement pris en compte et le champ de sélection de l'environnement est grisé.*

   ☛ *Pour en savoir plus sur les environnements et les utilisateurs, voir les chapitres **Gérer les environnements** et **Gérer les utilisateurs** du **HOPEX Administration - Supervisor**.*
3. Dans le champ **Identifiant**, saisissez votre identifiant.
4. Dans le champ **Mot de passe**, saisissez votre mot de passe (si nécessaire).

5. Cliquez sur **OK**.
   Lorsque vous êtes authentifié, la fenêtre de définition de votre espace de travail apparaît.



   Le champ **Personne** est défini automatiquement ; il indique le nom de la personne associée à l'identifiant renseigné dans la fenêtre de connexion.

6. Dans le champ **Référentiel**, sélectionnez votre référentiel de travail.

   ☛ *Le menu déroulant vous permet de faire apparaître la liste des référentiels disponibles dans l'environnement. Lorsque vous commencez à modéliser les données de votre entreprise, il est recommandé de créer un nouveau référentiel dans un nouvel environnement.*

   ☛ *Si vous n'avez accès qu'à un seul référentiel, celui–ci est automatiquement pris en compte.*

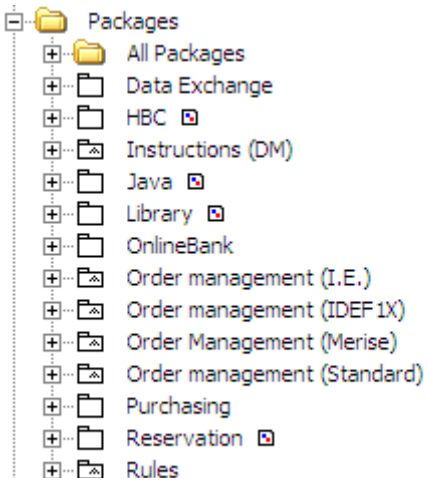
7. Dans le champ **Profil**, cliquez sur la flèche et sélectionnez le profil avec lequel vous voulez travailler.

   ☛ *Si vous n'avez qu'un profil, celui-ci est automatiquement pris en compte.*

8. Cliquez sur **OK**.
   Votre bureau apparaît.

Discovering the HOPEX UML Workspace

In the navigator, click on the "+" of "Packages" to display the first level list of packages.



The icon that appears alongside certain packages indicates that they are described by a diagram.

USE CASE DIAGRAM

The use case diagram constitutes a first step in description of an information system. It enables identification of the functionalities to be provided by the system to meet the requirements of organization actors; it therefore describes interactions between the system and the actors.

The following points are covered here:

- ✓ "Creating a Use Case Diagram", page 14
- ✓ "Use Case Diagram Elements", page 16

Creating a Use Case Diagram

A use case diagram is used to describe the interactions between the organization actors and the system, for each of the planned *use cases*.

> *A use case is a sequence of actions that produces an observable result for a particular actor. Scenarios illustrate use case examples.*

These use cases are grouped into *packages* representing the system boundaries.

> *A package partitions the domain and the associated work. It is used to organize elements into groups, particularly use cases and classes. Packages may be nested within other packages. Packages are linked by contractual relationships that define their interface.*

The package is the element from which you will create the use case diagram. However, for complex systems, you can also create this type of diagram from a use case in order to detail the latter.

Creating a Package

To create a package:

1. Open the **Main Objects** navigation window.
2. Right-click the **Packages** folder and select **New Package**.
   The Creation of Package dialog box appears.
3. Enter its **Name**.
4. Indicate the library and owner package if necessary.

   > *The default library is used to store an object if there is no current library at the time of its creation.*

5. Click **OK**.

The package is created and added to the list of packages.

> *When the **OK** or **Create** buttons are grayed, this is because the requirements for the dialog box in which they appear have not been completed.*

Creating the Use Case Diagram of a Package

To create a use case diagram:

1. Right-click the package concerned.
2. Select **New Diagram**.
   A dialog box opens indicating possible diagram types.
3. Select **Use Case Diagram**.
4. Check that the **Diagram initialization** check box is selected.

   > *The **Diagram initialization** option takes account of the object environment in the drawing.*

5. Click **Create**.
   The diagram opens in the **MEGA** Edit window. The frame of the package is positioned within the drawing.

Opening a Use Case Diagram

To open an existing use case diagram:

1. In the **Main Objects** navigation window, right-click the package that contains the use case.
2. Select **Use Case Diagram**.

The use case diagram is opened.

You are now in the drawing tool of **MEGA for UML**. The paragraphs that follow present objects of a use case diagram: actors, use cases, packages, etc. - as well as the links between these objects.

> ☛ *In the examples presented in this guide, object names may include spaces, upper case characters, and accented characters. It is important to note that if you have a generation tool using specifications created with **MEGA for UML**, and this tool is more restricted in the authorized characters and name lengths, it is preferable to adhere to the more restrictive rules in the **MEGA for UML** specification.*

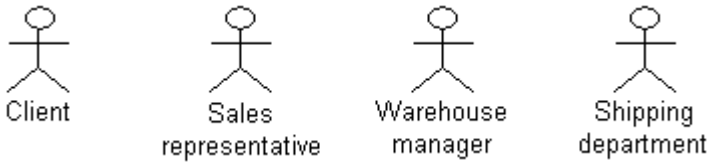
USE CASE DIAGRAM ELEMENTS

✓ "Actors", page 16
✓ "Use Cases", page 17
✓ "Packages", page 18
✓ "Participations", page 19
✓ "Associations Between Use Cases: Extension and Inclusion", page 20
✓ "Generalizations", page 23
✓ "Interfaces", page 24

Actors

An actor represents the role played by something or someone within the enterprise environment of the modeled system. It is related to the business activities of the enterprise, and interacts with the system in different use cases. It can be an element in the enterprise structure such as a division, a department, or a workstation.

Examples of actors:



To create an *actor* in a use case diagram:

1. In the diagram insert toolbar, click button **Actor (UML)**
2. Click in the diagram.
   The **Add Actor (UML)** dialog box opens.
3. Enter the name of the actor, "Receptionist" in this example.
4. Click the **Create** button.

   *If the name you enter corresponds to an actor that already exists in the repository, the **Create** button becomes **Connect**.*

   The actor then appears in the diagram.

   You can create several elements in a row without clicking in the toolbar each time, by double-clicking the button.

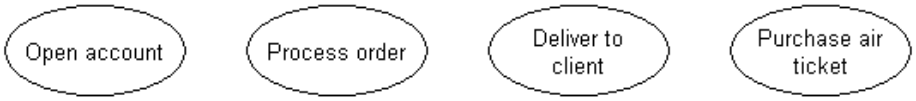
   To return to normal mode, press the <Esc> key or click on a different button in the toolbar, such as the arrow.

   *You can assign the same size to several items by selecting them and then applying the **Make Same Size** command in the **Drawing** menu. Follow the same steps to align objects: from the Edit menu, select **Align**. The last item selected is used as the reference. To select*

several items, click on each item in succession while holding down the Shift key.

Use Cases

Examples of *use cases*: processing an order, delivering to a client, opening an account, sending an invoice, establishing credit, purchasing an airline ticket, etc.



To create a use case in a diagram:

1. Click the **Use Case** button in the insert toolbar.
   The **Add Use Case** dialog box opens.
2. Enter the use case name and click the **Create** button.

The use case appears in the diagram.

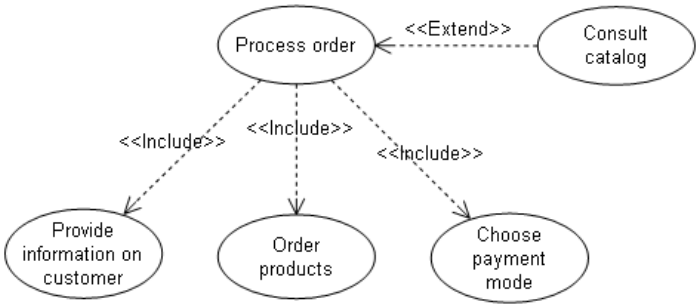
Zooming in on a use case

To open the diagram that describes a use case directly from the package diagram:

1. Right-click the use case.
2. Select **Use Case Diagram**.

☛ *You can also create a class diagram or object diagram using the command New.*

The use case diagram opens.

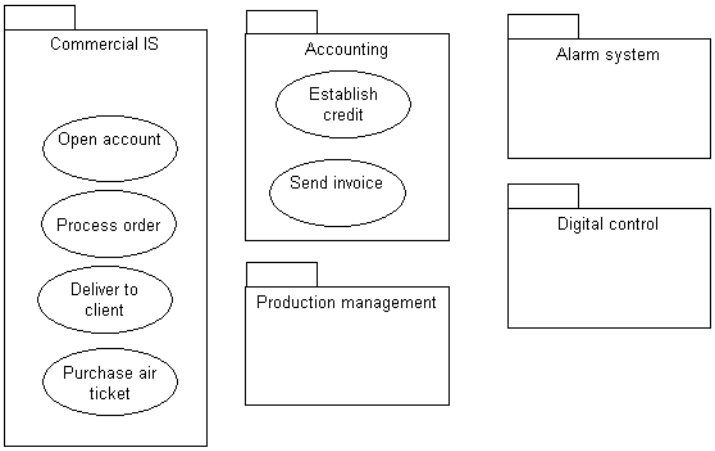


☺ *Zooming in on the description of an object is possible for all elements described by a diagram.*

Packages

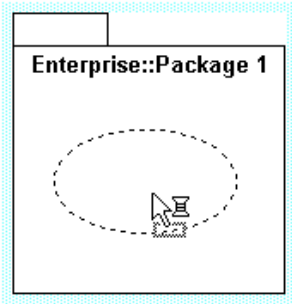
A package partitions the domain and the associated work. It is used to organize elements into groups, particularly use cases and classes. Packages may be nested within other packages. Packages are linked by contractual relationships that define their interface.

Examples of *packages*: the commercial information system, accounting, production management, digital control of a machine, an alarm system, etc.



You can create a package using the button in the toolbar. You can then increase its size in order to place use cases within it.

You can link a use case to a package simply by placing it within the package. When you have moved the object within the package, the package shape is highlighted to indicate that the object will be connected to it



*If the linked objects disappear under the package, click the package and select the **Send to Back** button in the Edit toolbar.*

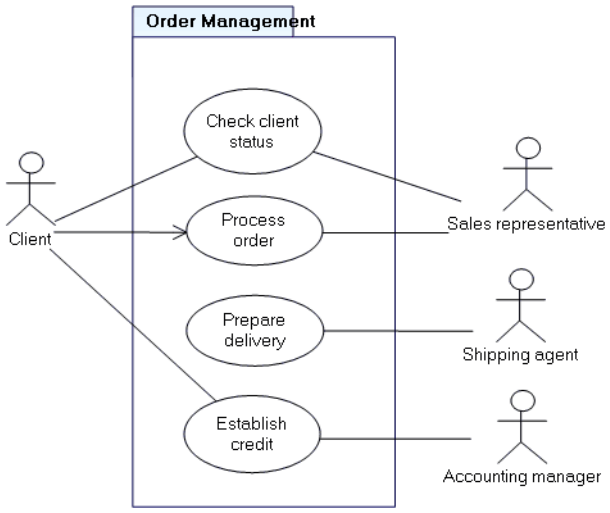
When you move a use case from one package to another using the mouse, it is unlinked from the first one and linked to the second. When you move it with the keyboard arrows, however, the links remain unchanged.

Participations

You can indicate which actor participates in each of the different use cases.

A participation indicates that an actor plays a role in a use case.

Examples of participation



- The sales representative participates in order processing and in checking client status;
- The shipping agent participates in delivery;
- The accounting manager participates in setting up loans, etc.

Creating participations

To create a *participation* in a use case diagram:

1. In the insert tool bar, click the **Participation** button
2. Click the actor concerned, and drag the mouse to the use case before releasing the button.
   A dialog box appears:
3. Enter the name of the participation and indicate if the actor is the initiator.

   *It is possible to specify the beginning of the use case by selecting the **IsInitiator** check box in the properties dialog box of the corresponding participation. An arrow appears on the line representing the participation.*
4. Click **OK**.

The link representing the participation appears in the diagram.

A participation is represented by a link, but it is in fact an object with its own properties.

The spool is not used to create participations. It is used to create certain types of links, such as those between packages and other objects.

*If you make a mistake, you can delete an object by right-clicking it and selecting the **Delete** command in its pop-up menu. You can also delete a link by right clicking on it and selecting **Delete** or **Disconnect** from the link pop-up menu.*

Multiplicities of a participation

Multiplicity can be specified on a participation:

- From the actor, to indicate that several instances of the actor participate in the same instance of the use case (example: participants in a meeting).
- From the use case, to indicate that the same instance of the actor participates in several instances of the use case (example: a sales representative processes several orders form the same customer).

To define multiplicities on a participation:

1. Right-click the participation and select **Properties**.
2. Click the **Characteristics** tab.
   A first frame allows you to define multiplicity of the actor, a second frame that of the use case.

Having been defined, the multiplicities appear in the diagram.

Associations Between Use Cases: Extension and Inclusion

When the system to be described is large, it is useful to have modeling mechanisms that can be adapted to the desired level of detail. Associations between *use cases* provide this ability.

When a use case includes too many alternatives and exceptions, these are represented separately as relationships that extend the standard use case.

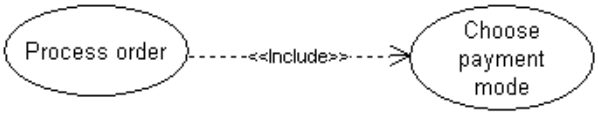
Inclusion relationship

One use case can be called automatically following another, for example validation of an order necessarily includes selection of a means of payment.

To indicate that one use case includes another:

1. In the use case diagram, click the **Uses Use Case** button
2. Click the use case, for example “Process Order" and drag the mouse to the case used, for example “Choose Payment Mode" before releasing the mouse button.

The link appears in the diagram, labeled “Include”.



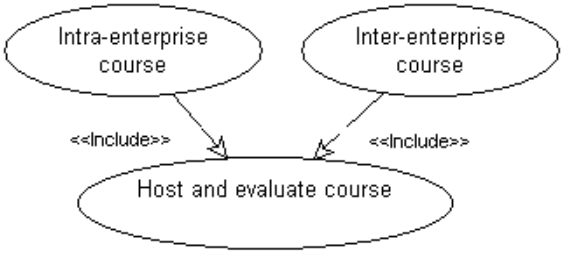
Examples of inclusion

In a training establishment, the following use cases:

- Inter-enterprise course (where the participants come from several different companies)
- Intra-enterprise course (where the participants all come from the same company)

can both include the following use case:

- Host and evaluate the course



In a company doing direct sales, the use case:

- Place an order

can reuse the following use cases:

- Provide client information
- Place a manufacturing order
- Propose a payment method

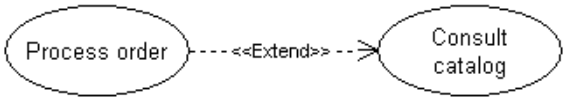
Extend Relation

One use case can result in execution of another. Unlike inclusion, which is automatic, extension is optional.

To indicate that one use case is an extension of another:

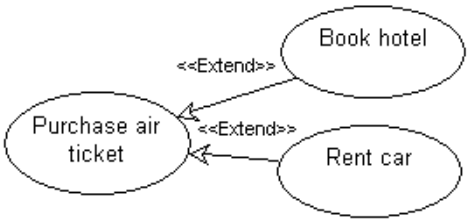
1. Click the **Extension** button
2. Click the use case, for example "Consult Catalog" and drag the mouse to the extension case, for example "Process Order" before releasing the mouse button.
   The Creation of Extension dialog box appears. You can define a constraint or an extension location.
3. Click **Finish**.

The link appears in the diagram, labeled “Extend”.



Extension example

The purchase of an airline ticket can also include booking a hotel room or renting a car.

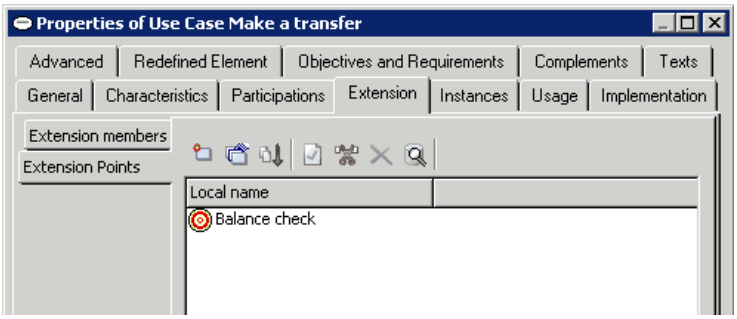


Extension point

The extension can intervene at a precise point in the extension case. This point is called the extension point.

To create an extension point on the extension case:

1. Open the properties dialog box of the extension use case.
2. Select the **Extension** tab and then the **Extension Points** subtab.
3. Click the **New** button.
   The extension point appears in the properties dialog box. You can rename it.

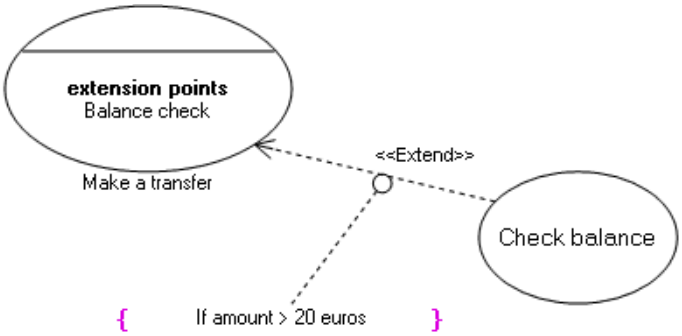


An extension point can be associated with a *constraint* which indicates the moment at which the extension intervenes. You can add a constraint at creation of the extension or later, in the extension properties dialog box.

A constraint is a declaration that establishes a restriction or business rule that must be applied on execution of processing.

Extension point example

The following example presents the use case of a bank transfer; above a sum of 20 euros, customer credit check is triggered.

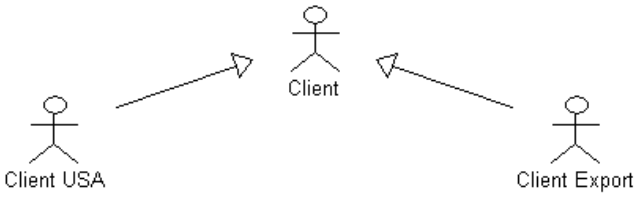


Generalizations

> *A generalization represents an inheritance relationship between a general class and a more specific class. The more specific class is fully consistent with the more general class and inherits its characteristics and behavior. However, it also contains additional information. Any instance of the more specific class is also an instance of the general class.*

The concept of *generalization* was initially used for classes, but has been extended to other UML concepts such as actor and use case.

Examples of generalizations between actors:

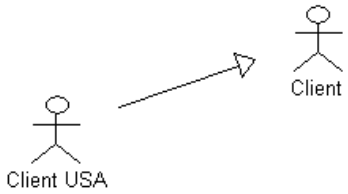


The "Client" actor can have the USA or Export specialization.

To create a generalization between actors:

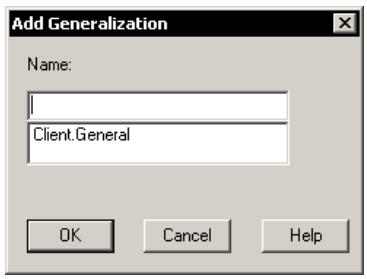
- Click the button and drag the link from the specialized actor (eg: USA client) to the more general actor (eg: Client).

The generalization then appears in the drawing.



☛ *In the same way you can create a generalization between two use cases.*

When creating a second generalization, a dialog box allows you to reuse the existing generalization if it involves the same subject.



Interfaces

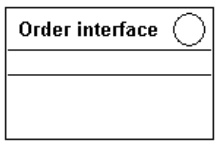
It is possible to complement the description of a use case or actor by describing the *interfaces* by which it communicates with its environment.

To create an interface:

1. Click the **Interface** button in the insert toolbar.

   ☛ *If the **Interface** button does not appear in the toolbar, select **View Views and Details** and select **Classes**.*
2. Click in the diagram.
3. In the dialog box that appears, enter the name of the interface and click the **Create** button.

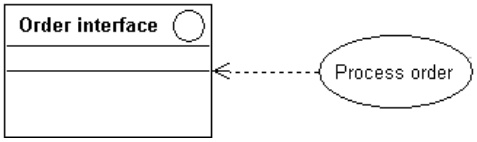
The interface appears in the diagram.



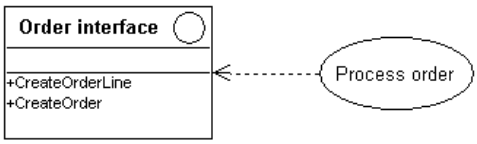
To specify the interface **supported** by a use case:

- Click the button and drag the link from the use case (eg: Process Order) to the interface (eg: Order Interface).

The link then appears in the diagram.



You can detail which operations the use case can carry out via this interface.



To indicate the interface **required** by a use case:

- Click the button and draw the link between the use case and the interface.

✓ For more details on required and supported interfaces, see "Linking interfaces to other objects", page 94.

THE CLASS DIAGRAM

A class diagram is used to represent the static structure of a system, particularly the types of objects manipulated in the system, their internal structure, and the relationships between them. An object diagram provides examples to illustrate a class diagram.

The class diagram specification is often considered the most important part in the modeling of an information system. The points covered in this chapter are:

- ✓ "Presentation of the Class Diagram", page 28
- ✓ "Creating a Class Diagram", page 30
- ✓ "Classes", page 31
- ✓ "Attributes", page 37
- ✓ "Operations", page 42
- ✓ "Signals", page 50
- ✓ "Associations", page 53
- ✓ "Generalizations", page 65
- ✓ "Specifying Interfaces", page 71
- ✓ "Specifying Dependencies", page 73
- ✓ "Specifying Parameterized Classes", page 74
- ✓ "Constraints", page 76
- ✓ "Object Diagram", page 77
- ✓ "The Class Editor", page 82
- ✓ "Generating a Class Diagram", page 86

Presentation of the Class Diagram

A class diagram is used to represent the static structure of a system, particularly the types of *objects* manipulated in the system, their internal structure, and the relationships between them.

> *An object is an entity with a well-defined boundary and identity that encapsulates state and behavior. Its state is represented by the values of its attributes and its relationships with other objects. Its behavior is represented by its operations and methods. An object is an instance of a class.*

Examples of objects:

- Business objects:
  - John Williams, Elizabeth Davis and Paul Smith are instances of the “person” class.
  - Orders 10533 and 7322 are instances of the "order" class.
  - SPD-1730 Monitor is an instance of the “item” class.
- Technical objects used for programming:
  - Dlg_Order_Create, Dlg_Client_Query are instances of the window class.
  - Str_Client_Name, Str_Product_Comment are instances of the "string" class.

Data modeling consists of identifying the classes representing the activity of the company, and defining the associations existing between them.

The classes and associations comprising the class diagram associated with a business area of the company must provide a complete semantic description.

In other words, one should be able to describe the activity of a company by using only these classes and associations.

This does not mean that each word or verb used in the explanation maps corresponds directly to an object in the class diagram. It means one must be able to state what is to be expressed, using these classes and associations.

The class diagram specification is often considered the most important part in the modeling of an information system.

An object diagram provides examples to illustrate a class diagram.

In particular, it is possible to illustrate a class diagram by showing the corresponding object diagram in the same drawing.

The Class Diagram: summary

In **HOPEX UML**, a class diagram includes:

- Classes, which represent the basic concepts (client, account, product, etc.).
- Associations, which define the relationships between the different classes.
- Attributes, which describe the characteristics of classes and, in certain cases, of associations.
- Operations, which can be executed on objects of the class.

The class diagram also contains multiplicity definitions.

See the glossary at the end of this guide for definitions of these and other concepts.

Creating a Class Diagram

A class diagram is created from a package.

To create a class diagram:

1. Right-click the package.
2. In the pop-up menu that appears, select **New** > **Diagram**.

☛ *The pop-up menu can be opened by clicking a package in the navigator, a folder on the desktop, or in a diagram that contains the package.*

3. In the dialog box that opens, select "Class Diagram" and click the **Create** button.

The new class diagram opens.

A class diagram can describe a package, a use case, a class, or an instance.

CLASSES

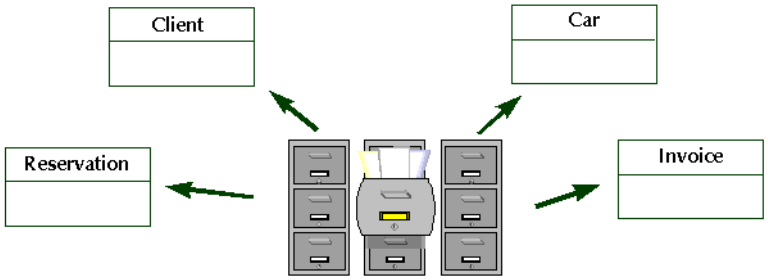
✓ "Definition: Class", page 31
✓ "Creating a Class", page 32
✓ "Class Properties", page 33
✓ "Class Stereotype", page 35

Definition: Class

A *class* is described by a list of attributes and operations.

A class is linked to other classes via *associations*. The set of classes and associations forms the core of a class diagram.

We can illustrate the class concept by comparing classes to index cards filed in drawers.



Classes can be technical objects used for programming.

`Examples: dialog box, rectangle, string, table, etc.`

Classes can represent technical objects used in industry.

`Examples: Alarm, Sensor, Zone`

Classes can also represent business objects:

`Examples: customer, order, product, person, company, etc.`

A class can also express a process, such as "Confirm client request", or implement a business rule, such as "Consistency in cost accounts".

☛ *See the glossary at the end of this guide for definitions of these and other concepts.*

Creating a Class

To create a class:

1. Click the **Class** button in the object insert toolbar.
2. Then click in the diagram workarea.
   The **Add Class** dialog box opens.
3. Enter the **Name** of the class and click the **Create** button.
   The class is then placed in the diagram.

> ☛ *In the examples presented in this guide, object names may include spaces, upper case characters, and accented characters. It is important to note that if you have a generation tool using specifications created with **HOPEX UML**, and this tool is more restricted in authorized characters and name lengths, it is preferable to adhere to the more restrictive rules of the **HOPEX UML** specification.*

You can create several classes successively without needing to click on the toolbar each time. To do so, double click the button.

To return to normal mode, click the arrow.

You can use the complete name of a class throughout by adding the name of the package to which it belongs to its name, separated by two colons.

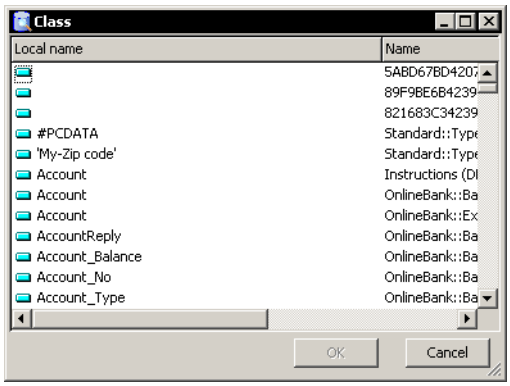
```
Example:
Enterprise::Sales Management::Client.
```

If one of the packages in the name does not exist, it is automatically created and linked to the class.

Finding an existing class

To find an existing class:

1. In the **Add Class** dialog box, select **List** in the drop-down list box using the arrow.
   The list of classes appears.



2. Select the desired class and click **OK**.
   The name of the selected class appears in the Add UML Class dialog box
3. Click **Add**.
   The class then appears in the drawing.

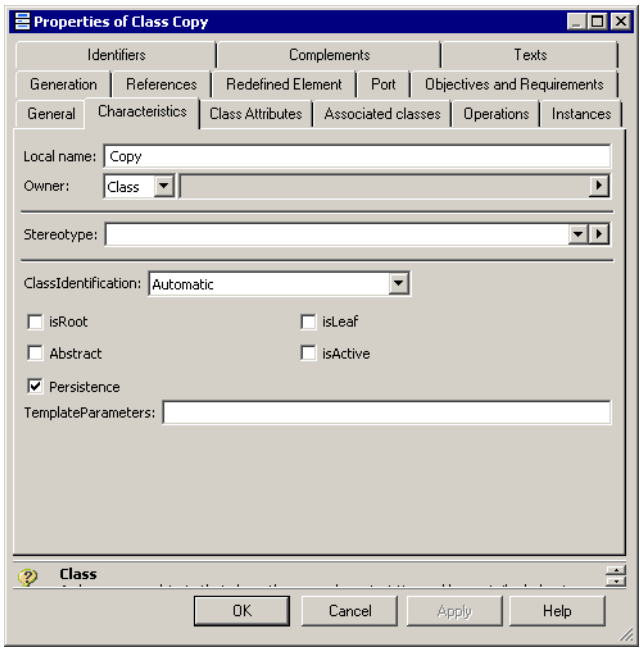
> ☛ *Note that you can also bring up the list of existing classes by pressing the Ctrl+L key.*
>
> *If the repository contains many classes, you can use the <Ctrl-Q> key search function to carry out more precise selection. See the **HOPEX Power Studio** guide for details on the use of this functionality.*

Class Properties

The properties displayed depend on the class stereotype.

To open the Properties dialog box of a class:

- Right-click the class and select **Properties**.
  It contains several tabs where you can define the class properties.



> ☛ *The comment at the bottom of the dialog box describes the field selected.*

characteristics tab

The **Characteristics** tab is used to enter or modify the different class characteristics:

- Its **Name**, which you can modify.

  ☺ *You can also modify the name of a class by clicking directly on the name in the drawing.*

- The *Package* to which the class belongs.
- The **Visibility** of the class as related to its package:
  - "Public": the class is visible to any element outside the package. This is the default visibility.
  - "Protected": the class is visible to elements that inherit it or have import dependencies with it, and to friends.
  - "Private": the class is only visible to elements that have import dependencies with it and to friends.
  - "Not specified".

    🕮 *Friends of a class are the classes that are authorized to access its internals. It is possible to specify the friends of a class in the complements tab of the properties dialog box of the class.*

- Its *stereotype*: see "Class Stereotype", page 35.

The other characteristics you can specify are the abstraction, persistence, and activity:

- If the class Is **Abstract**, it has no instances. It is only used to group operations or attributes common to its subclasses.
- **Persistence** specifies whether the objects in the class need to exist after the process or thread that created them, or whether they only last as long as the processing.
- Instances of a class which **Is Active** are able to trigger control flows without user intervention.

  `Example: An instance of the printer class can send an "Out of paper" message to the network administrator screen.`

- An **IsRoot** class is a class that has no superclasses in the tree of class generalizations.
- An **IsLeaf** class is a class that has no subclasses in the tree of class generalizations.

You can also specify the Parameters of a parameterized class (for C++).

☛ *See "To specify a parameterized class:", page 74 for further information.*

General tab

Certain general properties appear in all diagram elements.

- The name of the element creator.
- The name of the person who last modified the element.
- The modification status of the element, which indicates whether you have the right to modify the properties of this element when authorizations have been set up.

Texts tab

The **Comment** text allows you to comment a class.

> ☺ *Comments can be used to add key information to diagrams when certain details cannot be displayed in the drawing. These comments are included in the document describing the class diagram.*

Other tabs

Other tabs allow you to define or view:

- Attributes of a class (see "Attributes", page 37)
- Operations of a class (see "Operations", page 42)
- Associated classes (see "Associations", page 53)
- Instances of a class (see "Object Diagram", page 77)

Class Stereotype

A stereotype is a type of element that extends the semantics of the metamodel. Stereotypes must be based on existing types or classes whose structure they use. Other stereotypes can be created by the user.

Stereotypes available for a class are:

- **Org-Unit**: represents the role played by something or someone within the enterprise environment of the modeled system.
- **Auxiliary**:class that supports another central or fundamental class, generally by implementing a secondary logic or a control flow.
- **Implementation class**: is used to characterize the classes needed for physical implementation of the system.
- **Metaclass**: class of which the instances are themselves classes. As a general rule, metaclasses are used to build metamodels.
- **Control**: is used for classes that perform processing internal to the system. These generally require contributions from several classes.
- **Entity**: enables description of classes that are passive; that is that do not initiate interactions on their own. An entity object may participate in many different use cases and usually outlives any single interaction. They represent objects shared between the different actors who manipulate them.
- **Enumeration**: datatype containing a list of tabulated values.
- **Expression**: expressions of complex datatypes based on types.
- **Focus**: class that defines the main logic or control flow for the auxiliary class(es) that support it.
- **Boundary**: used to describe classes that are in direct contact with the system environment. Man-machine interfaces are of this type.
- **Interface**: an interface is a named set of operations that describe the behavior of an element. In particular, an interface represents the visible part of a class or package in a contractual client-supplier type relationship.

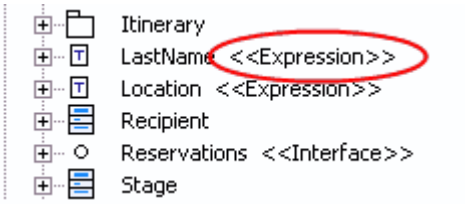
  > ☛ *These are interfaces between the different components of the computer system. These are not interfaces with system users, as those are considered boundary stereotypes. See "Specifying Interfaces", page*

71 for further information.

- **Worker**: represents a human actor who interacts with the system. A worker interacts with other workers and manipulates entities while participating in use case realizations.
- **Case worker**: a case worker interacts directly with actors outside the system.
- **Internal worker**: an internal worker interacts with other workers and other entities within the system.
- **PowerType**: metatype of which instances are sub-types of another type.
- **Structure**: class that describes a structure used in the programs.
- **Thread**: stereotype used in implementation of an active object as a light business process.
- **Primitive Type**: used to describe the datatypes.
- **Utility**: a class of this stereotype groups global variables and procedures useful for programming, and described as attributes and operations of this class.
- **Schema group**: class describing a type of XML element, the sub-elements of which form a group.
- **XML Document Definition Root**: class that describes the structure of a message exchanged between two systems using the XML language syntax.

Stereotype display option

An option allows you to display stereotypes in the navigation window of objects.



To activate this option:

1. From the **HOPEX** workspace, select **Tools** > **Options**.
2. In the left pane of the options window, select the **Workspace** folder.
3. In the right pane, select the option **Display stereotype of UML objects in navigator**.
4. Click **OK**.

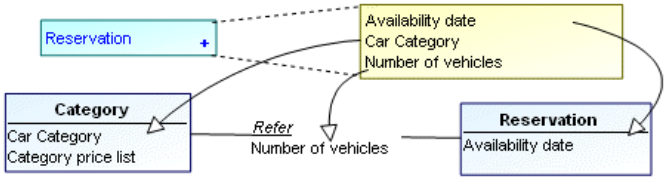
ATTRIBUTES

✓ "Definition: Attribute", page 37
✓ "Specifying Class Attributes", page 38
✓ "Attribute Properties", page 40

Definition: Attribute

An attribute is a named property of a class. This is the most basic data saved in the enterprise information system.

```
Examples:
"Client Name" (attribute of the client class).
"Client No." (identifier of the client class).
"Account balance" (attribute of the account class).
```

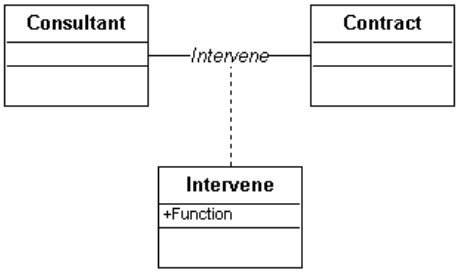


Classes and association classes may be characterized by attributes.

These attributes can be found by studying the content of messages circulating within the enterprise.

An attribute characterizes an association when its value depends on all the classes participating in this association.

In the diagram below, the "Role" that a "Consultant" plays in a "Contract" depends on the consultant and on the contract, and therefore on the "Intervene" association.



Specifying Class Attributes

To define class attributes:

1. Right-click the class.
2. Select **Attributes** in the pop-up menu.

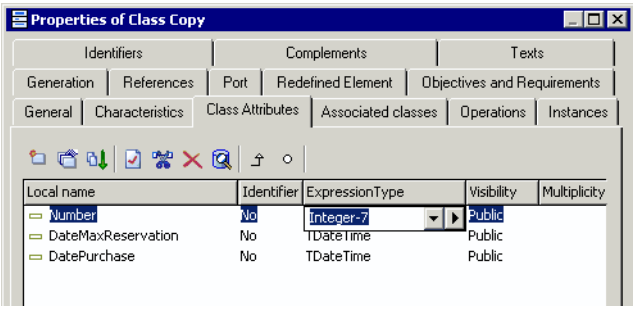
The attributes of the class are listed. It is possible to specify for each attribute:

- Its **Type**, which can be expressed as an expression.

  `Example: Integer.`

  ☛ *The expression must comply with UML syntax. See "Operation or Signal Signatures", page 45 for further information.*

To consult the list of possible types, click in the corresponding field and click the arrow to open the drop-down list box.



The list contains the types belonging to the different classes in the starting package (in this case, "Library").

To query other types, click the ▶ arrow on the right of the drop-down menu and select **Query**. The **Query** dialog box opens.

You may also enter the attribute type directly in the field.

- Its **Visibility**:
  - "Public": this is the default visibility. The attribute is visible to all.
  - "Protected": the attribute is visible to those inheriting its package, or to its friends.
  - "Private": the attribute is visible to its class or to its *friends*.

    Friends of a class are the classes that are authorized to access its internals. It is possible to specify the friends of a class in the complements tab of the properties dialog box of the class.
- Its **Multiplicity**, which is the number of times this attribute can be repeated in the class.

You can use the buttons in the toolbar to do the following:

Create a new attribute or a new element in general.

Reorder elements.

Display the properties of an element.

Delete an element.

Explore an element.

Inherited attributes

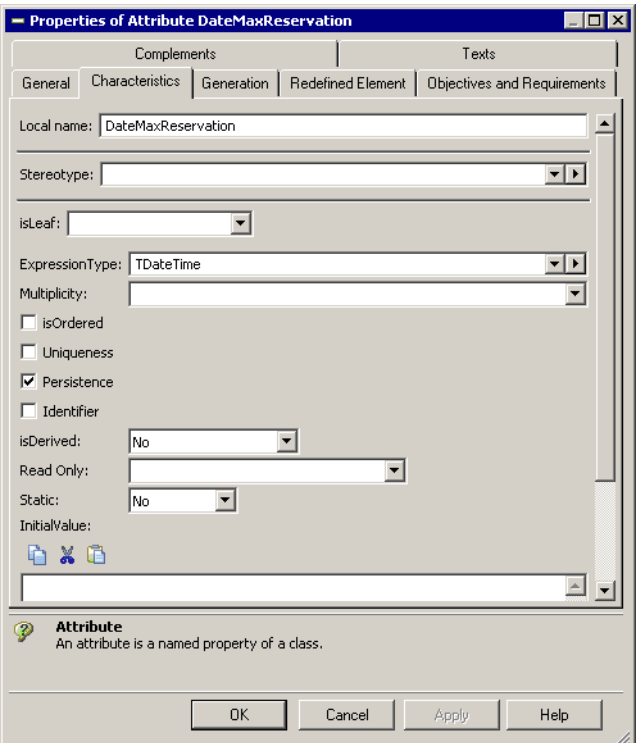
When a generalization exists between a general class and a more specialized class, the specialized class inherits the attributes of the general class.

- Click the **Inherited Attributes** button to view attributes inherited from other classes.

Attribute Properties

To open the Properties dialog box of an attribute:

❱ Right-click on the attribute and select **Properties**.



In this dialog box you can specify:

- The **Type** of the attribute in the form of an **Expression** (see "Types", page 41).
- Whether it is a **Static** attribute: specifies if the attribute can take specific values for each instance of the class or take one value characterizing the entire class.
  - "Yes": the attribute has a value that characterizes the entire class. The attribute "Telephone number length" for the "USA Client" class is 10 digits.
  - "No": the attribute can take a different value for each class instance. For example, the "Telephone number" attribute has a different value for each instance of the "Client" class.
- If the attribute has **Persistence**, specifying whether its value needs to exist after the process or thread that created it, or whether it only lasts as long as the processing.
- Its **Multiplicity**, which is the number of times this attribute can be repeated in the class.
- Whether it is **Read Only**, that is if its value can be modified once it has been specified.

- Whether it is a **Calculated Attribute**, specifying if its value is determined from the value of one or more other attributes.
- The **Initial Value** of the attribute, assigned when an instance of the class is created.
- Its **Visibility**:
  - "Public": this is the default visibility. The attribute is visible to all.
  - "Protected": the attribute is visible to those inheriting its package, or to its friends.
  - "Private": the attribute is visible to its class or to its friends.

Types

A type is used to group characteristics common to several attributes. Types are implemented as classes.

Any class can be used to type an attribute or parameter.

```
Example: Client, Order, Window, Table.
```

Classes of the "Primitive type" stereotype are created only for typing attributes or parameters. They are fixed.

```
Examples of primitive types:

String.

Integer.

Export address.

Monetary amount.
```

You can list existing types with the ▼ arrow, or create new types with the ▶ arrow.

The listed types include the classes in the package, used by the package, and in or used by packages of which it is the client.

> ☛ *To specify the structure of a type, place the corresponding class in the same diagram or in another diagram, and select the Properties command in its pop-up menu.*

Additional information on the use of primitive types is provided in the appendix to this guide.

OPERATIONS

- ✓ "Definition of an Operation", page 42
- ✓ "Specifying Class Operations", page 42
- ✓ "Operation Properties", page 44
- ✓ "Operation or Signal Signatures", page 45
- ✓ "Operation Parameters", page 47
- ✓ "Operation Methods (opaque behavior)", page 47
- ✓ "Object Diagram", page 77
- ✓ "Operation Exceptions", page 48
- ✓ "Displaying Class Attributes and Operations", page 48

Definition of an Operation

An operation is a service that can be requested from an object to affect a defined behavior. An operation has a signature, which may be used to specify the parameters it requires.

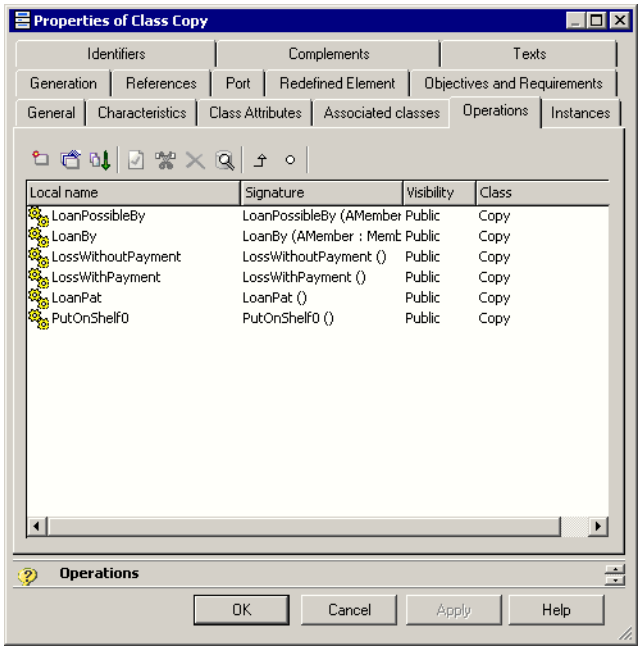
```
Examples:
"Age Calculation" (operation of the client class).
"Print" (operation of the drawing class).
"Calculate due dates" (operation of the loan class).
```

Specifying Class Operations

To specify class operations:

- Right-click the class and select **Operations**.

A dialog box presenting the class operations opens.



You can add operations by clicking the button and specifying their signatures.

Inherited operations

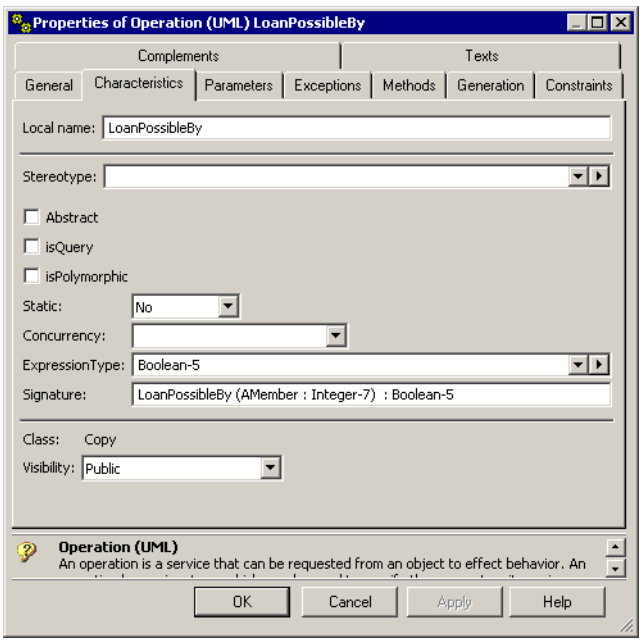
When a generalization exists between a general class and a more specialized class, the specialized class inherits the operations of the general class.

- Click the **Inherited Operations** button to view operations inherited from other classes.

Operation Properties

To open the Properties dialog box of an operation:

❯ Right-click the class and select **Properties**.



You can indicate for each operation:

- Its **Stereotype** to specify its use:
  - **Constructor**: creates an instance of the class.
  - **Destructor**: destroys an instance of the class.
  - **Iterator**: iterates through all instances of the class.
  - **Selector**: selects certain instances of the class.
- Whether it is a **Static** operation: if the operation can take specific values for each instance of the class or take one value characterizing the entire class.
- The **Concurrency**, to specify how the operation behaves when it is called several times simultaneously.
  - **Concurrency**: the operation responds simultaneously to the different calls.
  - **Protected**: the operation answers the first call and rejects ensuing ones.
  - **Sequential**: the operation responds successively to each call.
- If it is an **Is Query** operation, indicating that the object state is not modified.
- If the operation **Is Polymorphic**, to enable methods for this operation to be redefined in the subclasses.
- Its **Visibility**:
  - **Public**: this is the default visibility. The operation is visible to all.

- **Protected**: the operation is visible to elements inheriting its package or to its friends.
- **Private**: the operation is visible to its class or to its friends.

The following indications are used to further describe the operation signature.

- The **Expression type** of the operation.

  The operation expression type specifies the type of variable returned by the operation at the end of its execution.

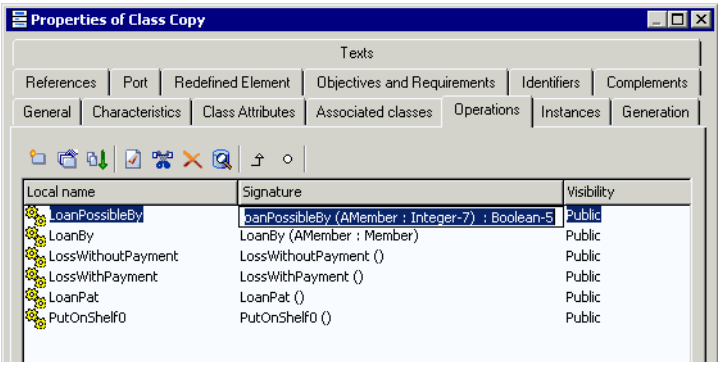
- Its **Signature**.

Operation or Signal Signatures

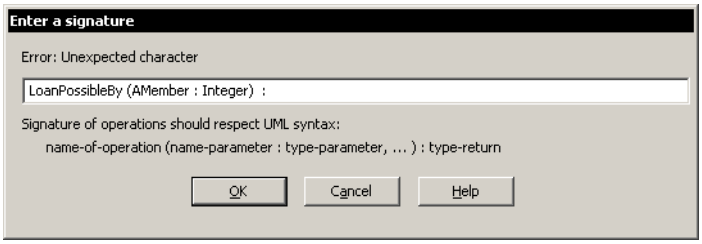
An operation or signal signature consists of the name of the operation (or signal), its return type, and its parameters with their types. Standard UML syntax is used for signatures, in the form: Ope0 (Param0: M-Bool): M-Bool.

The signature can be defined:

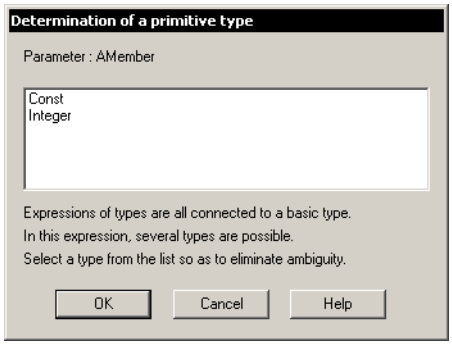
- Either in the properties dialog box of the operation or signal.
- Or in the Properties dialog box of the class to which the operation belongs, by directly entering the **Signature** in the **Operations** tab.



When the signature is entered directly, its validity is checked. If there is an error, a dialog box specifies the nature of the error and allows you to correct it.



When the signature contains a type preceded or followed by a complement (for example, Ope1(param1: const double), it is necessary to remove the ambiguity by selecting one of the words in the expression.



If the type does not exist, it is automatically created.

☛ *It is possible to filter out words that never correspond to a type, by instantiating an object of type "_UML ReservedWord" and naming it with the word to be filtered (in advanced metamodel: Tools > Options: Repository).*

The saved signature includes a reference to the type. If the type is renamed, the signatures that use it will reflect this change.

Signature syntax

The standard syntax for signatures is:

```
operationname(parameter1:typeexpression1,parameter2:typeexp
ression2,...):returnexpressiontype
```

Names containing spaces or special characters must be enclosed in single quotes ('Client name'). When a name contains an apostrophe, the apostrophe must be typed twice: 'Buyer''s Name'

```
Examples of signatures:
Unstock (Product0: Integer(3), Quantity0: Integer): Boolean
'Create order' ('Client name' : Client): Byref Variable
```

In a signature specification, it is possible to specify the package to which a class belongs, followed by two colons.

```
Example: Enterprise::'Sales Management'::Client.
```

The listed class is linked to the parameter or return type. If it does not exist, it is created. Any packages listed in the path that do not exist are also created, and linked to the class.

If the package is not specified, a dialog box will enable you to select from similarly named classes.

Operation Parameters

In the Properties dialog box of an operation, the **Parameters** tab allows you to specify:

- The operation **ExpressionType**, eg. Integer(5).
- Its **defaultValue**, eg. 0.
- Its **Direction**: at input and/or output of the operation.

To create a new *parameter*:

- Right click in the window and select **New**.

> *A parameter is the specification of a variable, which can be modified, sent or returned. A parameter can specify a name, a type and a direction. Parameters are used for operations, messages and events.*

> *An argument is a specific value corresponding to a parameter.*

Operation Methods (opaque behavior)

A method - or opaque behavior - is a textual representation of implementation of an operation, class or component. It specifies the algorithm or procedure that produces results of an operation or behavior of an element.

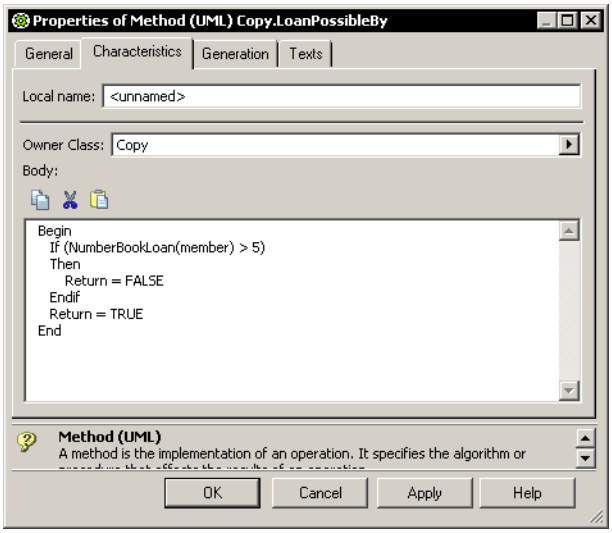
To define the method implementing an operation:

1. Open the Properties dialog box of the operation.
2. In the **Methods** tab, click the **New** button.
3. Click in the **Owner Class** column and select **Candidate Operation Classes**.
4. The tool searches for classes that could supply implementation methods. You can also create a new owner class of the method.

To enter the body of the text and the method that implements the operation:

1. Open the properties dialog box of the method.
2. Select the **Characteristics** subtab.

3. Define the method in the **Body** frame.



When a class has several subclasses, each subclass can perform the operation using a different method.

The **Method** tab presents the method relative to the selected class.

Operation Conditions

You can define operation conditions in the form of constraints.

The **Constraints** tab in the operation properties dialog box is used to specify:

- A **PreCondition** that must be met before the operation executes.
- The **Body**, that must be met at operation execution.
- A **PostCondition** that must be met after executing the operation.

If a condition is not respected, an exception is generated.

Operation Exceptions

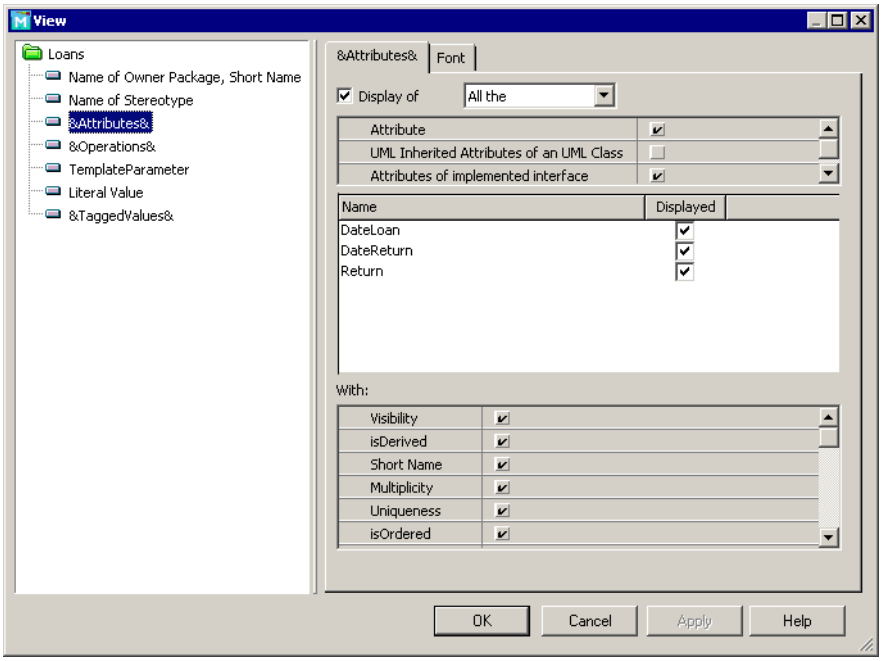
The **Exceptions** tab allows you to display error messages sent by the operation when an exception occurs and to specify their signatures.

Displaying Class Attributes and Operations

To modify how the attributes and operations for a class are displayed:

1. Right-click the classes whose attributes you want to display.

2. Select **Shapes and Details**.
   Use the Display dialog box to select what elements are to be displayed.
3. Select **Attribute** in the tree presented.



4. Select the attributes you want to see displayed.

You can display **All the** attributes, **Some of the** attributes (select them from the list), or **None of the** attributes.

You can request display of the **Visibility**, *Type*, … of each of the attributes.

A type is used to group characteristics shared by several attributes. Types are implemented as classes.

You can hide or show the compartment containing the class attributes in the drawing, by selecting or clearing the Display of check box.

Proceed in the same manner to indicate how operations are to be displayed, but instead, select **Operations** in the tree.

SIGNALS

Defining a Signal

A signal is a named event that can be explicitly invoked. Signals may have parameters. A signal can be sent to an object or a set of objects. It can be invoked as part of the participation of an actor in a use case.

A message can be sent or received by a class. It can also be sent by an operation after an exception.

Specifying Class Signals

Creating a sent or received signal

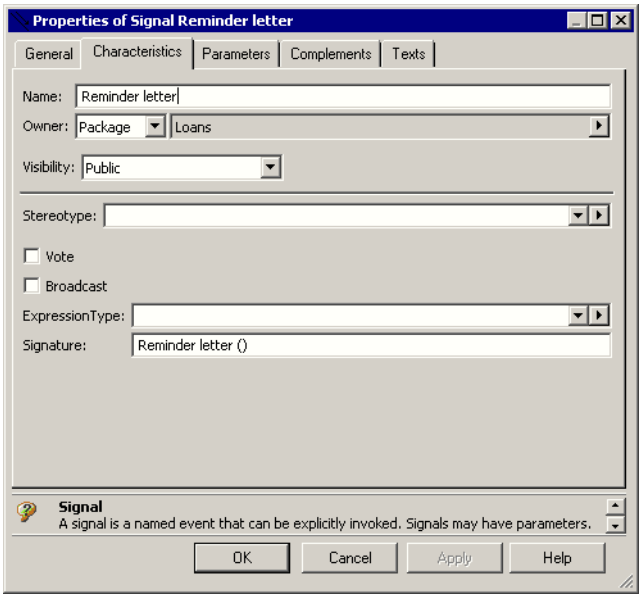
To specify what signals can be sent or received by a class:

1. Open the properties dialog box of a class.
2. Select the **Complements** tab.
3. In the menu tree presented, select **sentSignal** or **receivedSignal** then click the button.
4. Indicate the name of the signal and click **OK**.

Signal properties

To open the properties dialog box of a signal:

- In the properties dialog box of a class, in the **Complements** tab, right-click the signal and select **Properties**.
  The Properties dialog box of the signal appears.



You can indicate for a signal:

- Its **Stereotype** to specify its use:
  - **Exception**: an error signal is generated when an exception occurs during the execution of an operation.
- Its **Visibility** related to the package:
  - **Public**: this is the default visibility. The signal is visible to any element outside the package.
  - **Protected**: the signal is visible to inherited elements or friends.
  - **Private**: the signal is visible to its class or to its friends.

    Friends of a class are the classes that are authorized to access its internals. It is possible to specify the friends of a class in the complements tab of the properties dialog box of the class.
- The **ExpressionType** of the signal (see *expression type*)..

  The ExpressionType of a signal specifies the type of variable returned by the signal on its receipt by the addressee.

A signal can be a request to **Vote** sent to each active object, asking if it is possible to perform a specific action such as closing a Windows session.

A signal can be a general **Broadcast** to all active objects.

Signal parameters

The *Parameters* of the signal are specified in the **Parameters** tab of its Properties dialog box. You can specify:

- The operation **ExpressionType**, eg. Example: Integer(5).
- Its **Default Value**. Example: 0.
- Its **Direction**: at input and/or output of the operation.

A parameter is the specification of a variable, which can be modified, sent or returned. A parameter can specify a name, a type and a direction. Parameters are used for operations, messages and events.

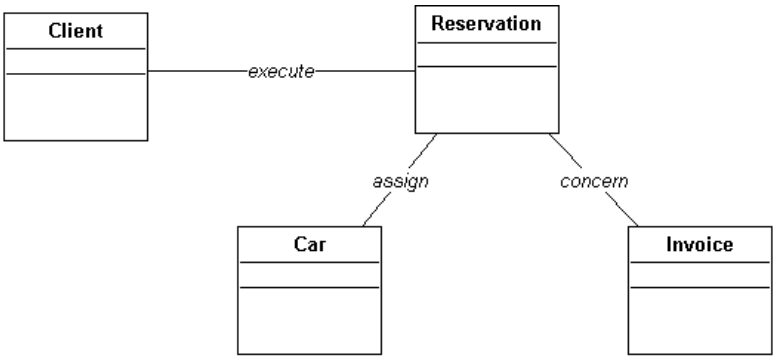
An argument is a specific value corresponding to a parameter.

ASSOCIATIONS

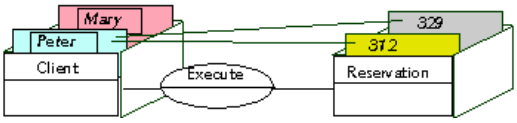
An association is a relationship that exists between two classes.

An association is binary when it links two classes, ternary when it links three classes, etc.

Associations can be compared to links between index cards.



The following drawing provides a three-dimensional view of the situations a class diagram can store.



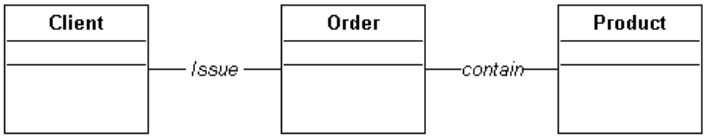
Peter and Mary are clients. Peter has made reservations numbers 312 and 329.

A class diagram should be able to store all situations in the context of the company.

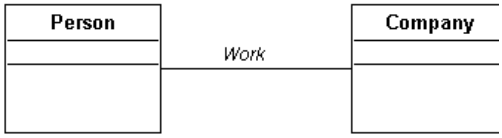
☛ *The diagram should not allow representing unrealistic or aberrant situations.*

Examples of associations:

- A client issues an order.
- An order includes several products.



- A person works for a company.



- An alarm is triggered by a sensor.

A sensor covers a zone.

A window displays a string of characters.

Creating an Association

To create an *association*:

1. Click the **Association** button in the objects toolbar.
2. Click one of the classes concerned and drag the mouse to the other class before releasing the button.
   The association is indicated by a line in the diagram.

If there is already an association between the two classes, the **Add an Association** dialog box opens. It enables creation of a new association or selection from existing associations.

You can specify a name for the association in its Properties dialog box, accessed from its pop-up menu.

☛ *If you make a mistake, you can delete an element or a link by right-clicking it and selecting the Delete command in the pop-up menu.*

You can also create an association from the class editor. See "Creating objects in the class editor", page 84.

Roles (or Association Ends)

It is possible to describe the different *roles* played by the classes in associations and to specify their multiplicity and their navigability.

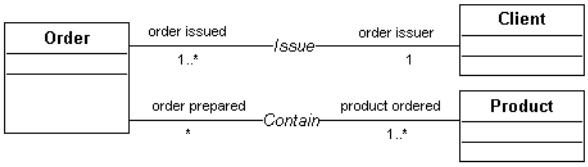
Each end of an association specifies the role played by the class in the association.

The role name is distinguished from the association name in the drawing by its position at the link end. In addition, the role name appears in a normal font, while the association name is italicized.

When two classes are linked by only one association, the name of the classes is often sufficient to describe the role. Role names are useful when several associations link the same two classes.

Examples of roles:

- A client is the *order issuer*.
- An order is *issued* by a client.
- An order is *prepared* from products.
- A product is *ordered*.



A person is an *employee* of a company.

A company is the *employer* of these persons.

An alarm is triggered by one or more sensors.

A zone is *covered* by a sensor.

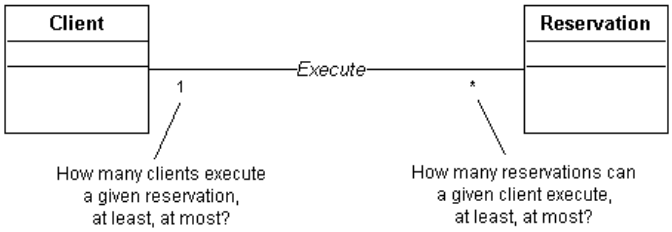
One or more strings are *displayed* in a window.

Multiplicity of a Role

*The multiplicity specifies the range of allowable cardinalities a set may assume. This is primarily indicated for each role that classes play in an association. It can assume the values *, 0..1, 1, 1..*, 2..*, 4..10, etc. The default value is *.*

The cardinality is the number of elements in a set.

The *multiplicity* expresses the minimum and maximum number of instances of a class that can be linked by the association to each instance of the other class.

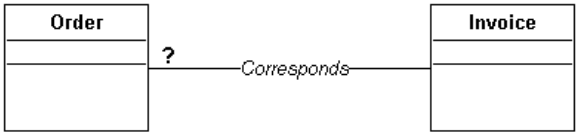


The usual multiplicities are "1", "0..1", "*" or "0..*", "1..*", and "M..N" where "M" and "N" are integers:

- The "1" multiplicity indicates that one and only one instance of the class is linked by this association to each instance of the other class.
- The "0..1" multiplicity indicates that at most one instance of the class can be linked by this association to each instance of the other class.
- The "*" or "0..*" multiplicity indicates that any number of instances of the class can be linked by the association to each instance of the other class.
- The "1..*" multiplicity indicates that at least one instance of the class is linked by the association to each instance of the other class.
- The "M..N" multiplicity indicates that at least M instances and at most N instances of the class are linked by the association to each instance of the other class.

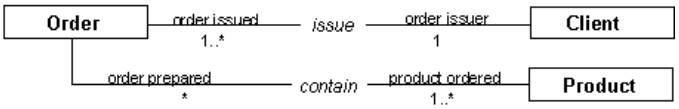
1	One and one only
0 / 1	Zero or one
M..N	From M to N (natural integer)
*	From zero to several
0..*	From zero to several
1..*	From one to several

The following example illustrates the significance of the different multiplicities:



- 0..1 : An order corresponds to zero or at most one invoice.
- * : No restriction is placed on the number of invoices corresponding to an order.
- 1, Each order has one and only one corresponding invoice.
- 1..* : Each order has one or more corresponding invoices.

Other examples of multiplicity:

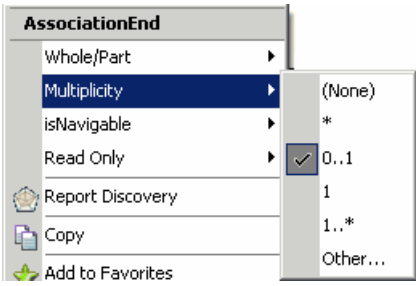


- 1..* : A client can issue one or more orders.
- 1, An order is issued by one and only one client.
- 1..* : An order contains one or more products.
- * : A product can be contained in any number of orders, including no orders.
- 0..1 : A person works for a company.
- 1..* : An alarm is triggered by one or more sensors.
- 1, A sensor covers one and only one zone.
- 1..* : A window displays one or more strings.

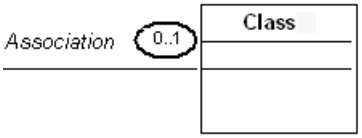
Specifying role multiplicity

To specify association end multiplicity:

❯ Right-click the line between the association and the class and select **Multiplicity**.



The multiplicity now appears on the role.



☛ *If the menu you see does not propose multiplicity, verify that you clicked on the line indicating the association end and not on the association.*

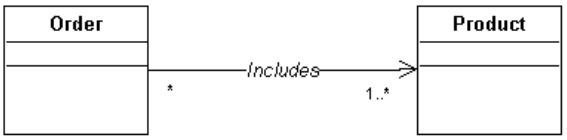
☺ *All the information specified with the pop-up menu can also be viewed and modified in the role properties dialog box.*

Association End Navigability

IsNavigable specifies in which direction(s) an association between two classes can be traversed. To avoid crowding the drawing, this is only indicated when only one direction is possible.

Example of navigability:

- It is important to be able to find out what products are contained in an order.
- However, it is rarely useful to be able to find all orders that concern a product.



Specifying navigability for a role

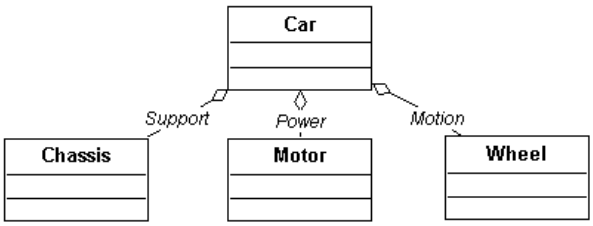
To indicate that an association is navigable in one direction only:

1. Right-click the non-navigable role.
2. Select **IsNavigable > No**.
   An arrow representing the navigability now appears for the other role.

Association End Aggregation

Aggregation is a special form of association, indicating that one of the classes contains the other.

```
Example: A car includes a chassis, an engine, and wheels.
```



Specifying role aggregation

To specify role aggregation:

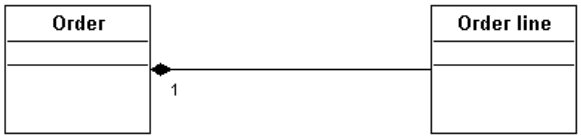
1. Right-click the role.
2. Select **Whole/Part** > **Aggregate**.
   A diamond now appears on the role, representing the aggregation.

Association End Composition

A composition is a strong aggregation where the lifetime of the components coincides with that of the composite. A composition is a fixed aggregation with a multiplicity of 1.

```
Example: An order consists of several order lines that will
no longer exist if the order is deleted.
```

Composition is indicated by a black diamond.

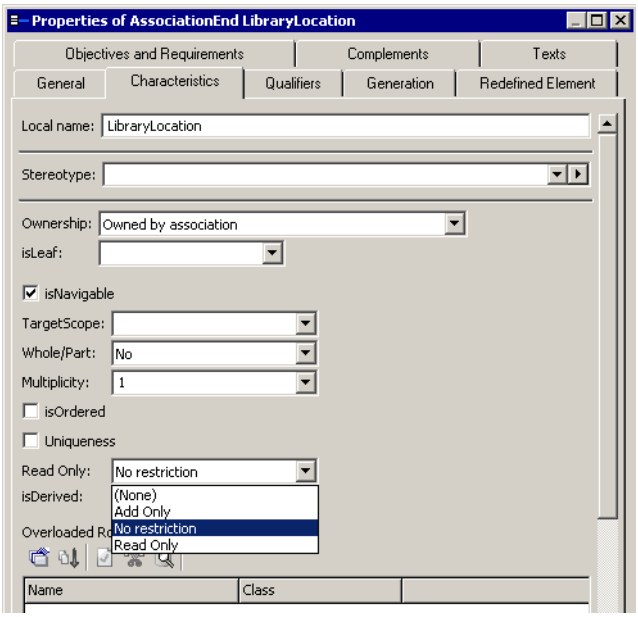


Role Changeability

Read Only specifies whether the role played by a class in an association may be modified after it has been created. By default, the role of a class in an association is considered changeable.

```
Example: An order includes an order line for each of the
ordered products. These order lines can no longer be
modified after the order has been saved.
```

You can indicate whether a role is changeable using the role pop-up menu or the role properties dialog box.



The **Read Only** characteristic of the role can have the following values:

- **Add only**: it is still possible to link new objects with this association, but already linked objects cannot be unlinked.
- **Read Only**: linked instances can no longer be unlinked. Nor is it is possible to add a new link.
- **No Restriction**: new instances can be linked or unlinked at any time with no constraints.

Role Order

It is possible to specify whether or not a role Is Ordered. For example, for a client order, it can be useful to store the sequence in which its lines appear.

To specify that a role is ordered:

1. Open the **Properties** dialog box of the role.
2. In the **Characteristics** tab, select the **IsOrdered** check box.

Role Static Property

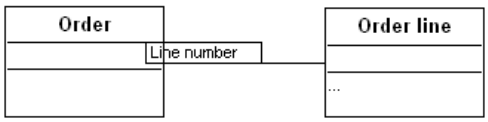
As for an attribute, it is possible to specify if a role can take specific values for each class instance, or take a value characterizing the entire class:

1. Open the properties dialog box of the role.
2. Click the **Characteristics** tab.
3. In the **Static** box, select:
   - "Yes": so that the role can take a value characterizing the entire class.
   - "No": so that the role can take a different value for each class instance.

Role Qualifier

A qualifier is an attribute whose values partition the set of objects related to an object across an association.

```
Example: An order includes several order lines. The order
line number can be used as the qualifier that identifies
each line.
```

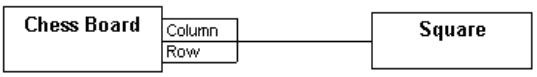


To specify a qualifier:

1. Right-click the role and select **Properties**.
   The Properties dialog box of the role opens.
2. Select the **Qualifiers** tab.
3. To add a new qualifier to a role, right-click in the dialog box.
4. In the pop-up menu that appears, select **New**. The same pop-up menu can then be used to modify the qualifier properties.

Several qualifiers may be needed to uniquely identify each object in a class.

For example, each square on a chessboard is identified by its row number and column number on the chessboard.



Overloading a Role

A role can inherit a role defined at higher level. Overloading enables definition of additional properties on an inherited role.

To overload a role:

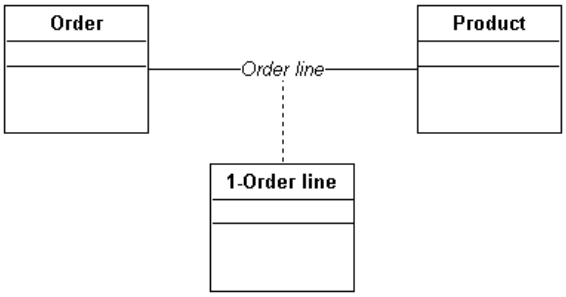
1. Open the properties dialog box of the role.
2. In its properties dialog box, click the **Characteristics** tab.
3. In the **Overloaded Role** frame, click the **Connect** button.
   The Query dialog box appears:
4. Select **Subseted properties**. This command displays roles that can be overloaded.
5. Select the role in question and click **OK**.

Association Classes

An association class is an association that also has class properties as attributes.

It is helpful to create an association class in order to specify the characteristics of an association.

```
For example, the quantity of the requested product needs to
be specified on each order line.
```



To create an association class:

1. Create a new class.

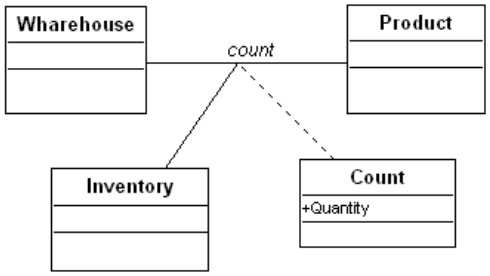
2. Using the **Link** button, create a link between the class and the association.
   The association class is linked to the association by a dotted line.

   ☛ *As for standard classes, it is possible to hide the compartments and resize the association class using the Display command in its pop-up menu.*

Displaying an N-ary Association

Certain associations associate more than two classes. These associations are generally rare.

```
Example: When taking inventory, a certain quantity of
product was counted in each warehouse.
```

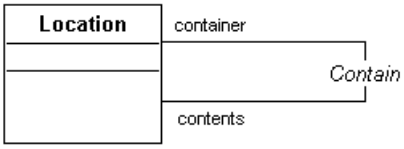


To create a ternary association:

1. First create the association between the two classes.
2. Click the **Association Role** button
3. Draw a link between the association and the third class.

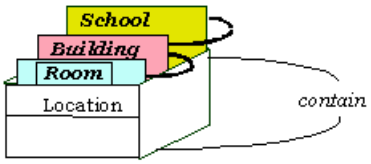
You can then proceed as described above to create an association class if needed.

Reflexive Associations



Certain associations use the same class several times.

A classroom, a building, and a school are all locations.



A classroom is contained in a building, which is contained in a school.

A reflexive association concerns the same class at each end.

Creating a reflexive association

To create a reflexive association:

1. Click the **Association** button in the toolbar:
2. Click on the class concerned and drag the mouse outside the class, then return to it and release the mouse button.
   The reflexive association appears in the form of a half-circle.

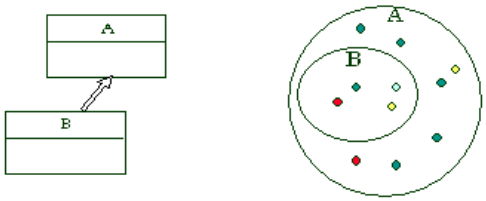
   ☛ *If there is an association of a class to itself, the roles need to be named in order to distinguish between the corresponding links in the drawing.*

GENERALIZATIONS

A generalization represents an inheritance relationship between a general class and a more specific class. The more specific class is fully consistent with the more general class and inherits its characteristics and behavior. However, it also contains additional information. Any instance of the more specific class is also an instance of the general class.

✓ "What is a Generalization?", page 65
✓ "Multiple Subclasses", page 67
✓ "Advantages of Subclasses", page 67
✓ "Multiple Inheritance", page 68
✓ "Creating a generalization", page 68
✓ "Discriminator", page 69

What is a Generalization?



Class A is a generalization of class B. This implies that all objects in class B are also objects in class A. In other words, B is a subset of A.

B is then the subclass and A the superclass.

Example A: Person, B: Bostonian.

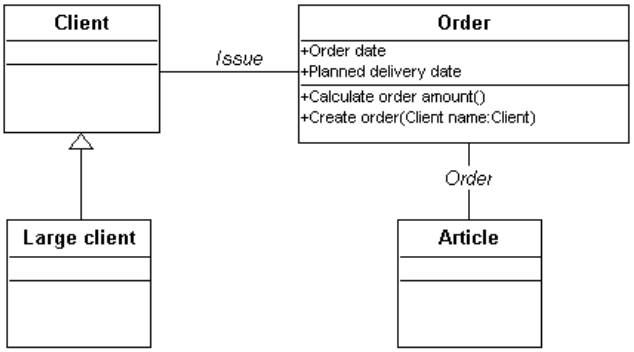
B is a subset of A, so the objects in class B inherit the characteristics of those in class A.

It is therefore unnecessary to redescribe for class B:

- Its attributes
- Its operations
- Its associations

Example

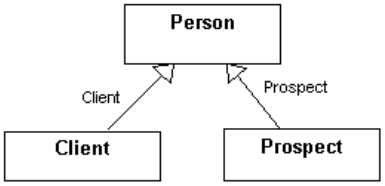
The "Large Client" class, representing Clients with a 12-month revenue exceeding $1 million, can be a specialization of the Client class (origin).



In the above example, the associations and attributes specified for “Client” are also valid for “Large client”.

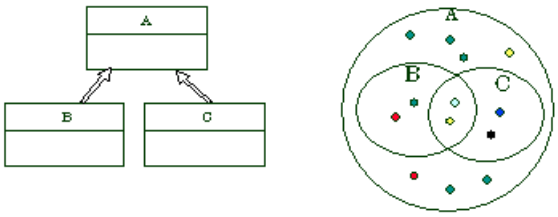
Other examples of generalizations:

- Prospect and client are two subclasses of "person".



- Export order is a subclass of the "order" class.
- Individual person and corporate person are two subclasses of the "person" class.
- Polygon, ellipse, and circle are subclasses of the "shape" class.
- Oak, elm, and birch are subclasses of the "tree" class.
- Motor vehicle, all-terrain vehicle, and amphibious vehicle are subclasses of the "vehicle" class.
- Truck is a subclass of the "motor vehicle" class.

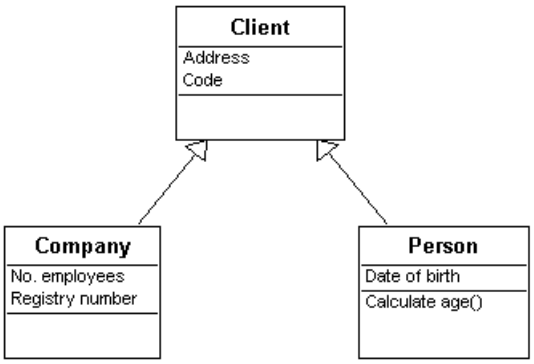
Multiple Subclasses



When a class has multiple subclasses, they:

- are not necessarily exclusive.
- do not necessarily partition the set.

Advantages of Subclasses

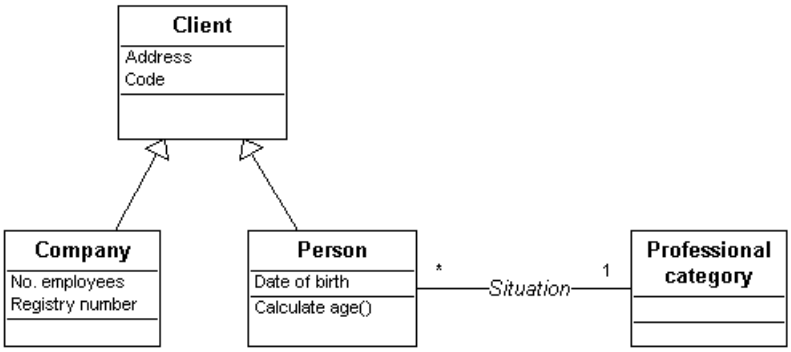


A subclass inherits all the attributes, operations, and associations of its superclass, but can have its own attributes or associations that the superclass does not have.

A subclass can also have specific attributes. These only have meaning for that particular subclass. In the above example:

- "Registry number" and "number of employees" only have meaning for a "company".
- "Date of birth" is a characteristic of a "person", not a "company".
- It is also useful to calculate the "age" of a "person". This attribute and this operation are generally not needed for a "company".

A subclass can also have specific *associations*.



- A "person" falls into a "socio-professional group": "manager", "employee", "shop keeper", "grower", etc. This classification makes no sense for a "company". There is also a classification for companies, but it differs from that for persons.

Multiple Inheritance

It is sometimes useful to specify that a class has several superclasses. The subclass inherits all the characteristics of both superclasses. This possibility should be used carefully.

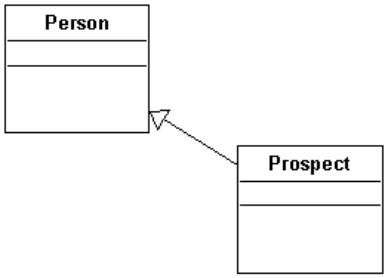
☛ *Multiple inheritance is not taken into account when generating tables.*

Creating a generalization

To create a generalization:

1. Click the **Generalization** button in the toolbar.
2. Click the subclass concerned, and drag the mouse to the superclass before releasing the button.

The generalization is now indicated in the diagram by an arrow.

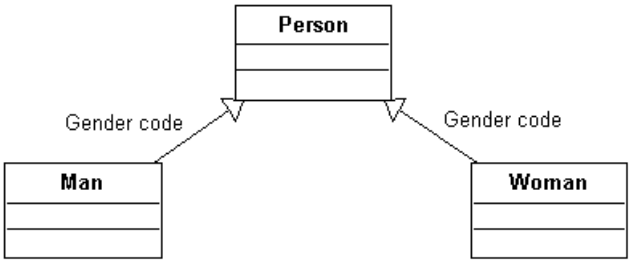


You can also create a generalization from the class editor. See "Creating objects in the class editor", page 84.

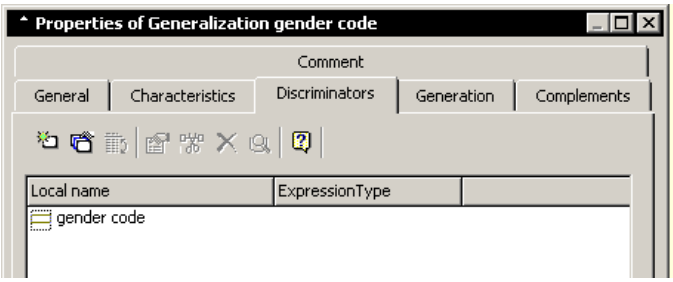
Discriminator

The discriminator is a generalization attribute whose value divides the objects into the subclasses associated with the generalization.

For example, the gender code attribute divides the objects in the person class into the man and woman subclasses.



You can define discriminator(s) in the generalization properties dialog box.



☛ *It is also possible to specify whether a generalization:*

Is Disjoint: An instance cannot belong to two subclasses of the generalization simultaneously.

Is Complete: All instances of the superclass belong to at least one of the subclasses of this generalization.

SPECIFYING INTERFACES

An interface represents the visible part of a class or package in a contractual client-supplier type relationship. The interface is a class stereotype.

An interface is a named set of operations that describe the behavior of an element. In particular, an interface represents the visible part of a class or package in a contractual client-supplier type relationship.

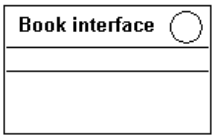
These are interfaces between the different components of the computer system.

Creating an Interface

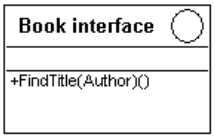
To create an interface class:

1. Click the **Interface** button in the toolbar and click in the drawing.
2. Enter its name in the dialog box that appears.

The interface class appears in the drawing with a distinctive circle top right:

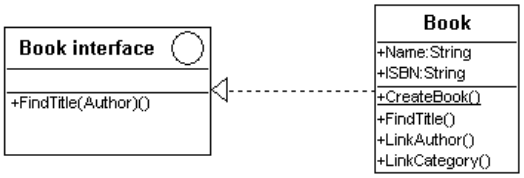


You can then specify the operations of the interface as for any other class.



To specify that an interface is supported by a class:

1. Click the button.
2. Then create the link from the supplier class to the supported interface.



To indicate that a class requires an interface:

1. Click the button.
2. Then create the link from the client class to the interface required.

SPECIFYING DEPENDENCIES

You can create a package using the button in the toolbar.

> ☛ *The **Views** button allows you to specify the buttons that you want to appear in the objects toolbar.*

To indicate that a package references a class or another package:

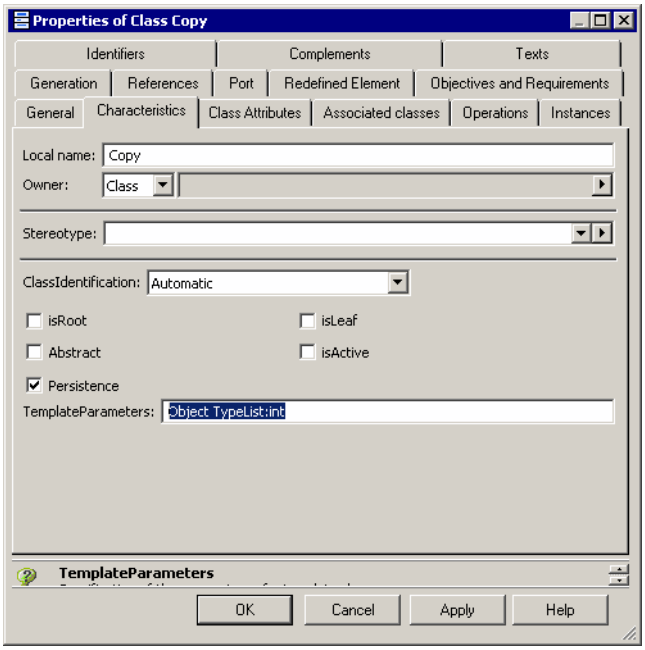
1. Click the button.
2. Then carry out the link from a package to the package or class that it references.

SPECIFYING PARAMETERIZED CLASSES

A parameterized class enables definition of characteristics and a behavior that varies as a function of the value of certain parameters. For example, a parameterized class can be used to manage object lists. In this case the parameter will be the object type to be managed in the form of a list. This type of class is implemented in particular in C++ language.

To specify a parameterized class:

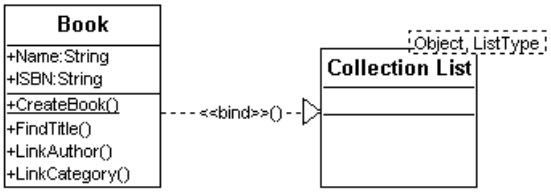
1. Open the **Properties** dialog box of the class and select the **Characteristics** tab.
2. You can enter the parameters and specify their type if necessary.



The class parameters are displayed at top right.

To link a class to a parameterized class:

- Click the button.
- Create the link from the class to the parameterized class.



CONSTRAINTS

A constraint is a declaration that establishes a restriction or business rule generally involving several classes.

Most constraints involve associations between classes.

Examples of constraints:

- The person in charge of a department must belong to the department.
- Any invoiced order must already have been delivered.
- The delivery date must be later than the order date.

A sensor covering a zone can trigger an alarm for that zone only.

To create a constraint:

1. Click the **Constraint** button in the object toolbar.

   ☛ *If it is not displayed, select **View Views and Details** and select the "Constraints" check box.*

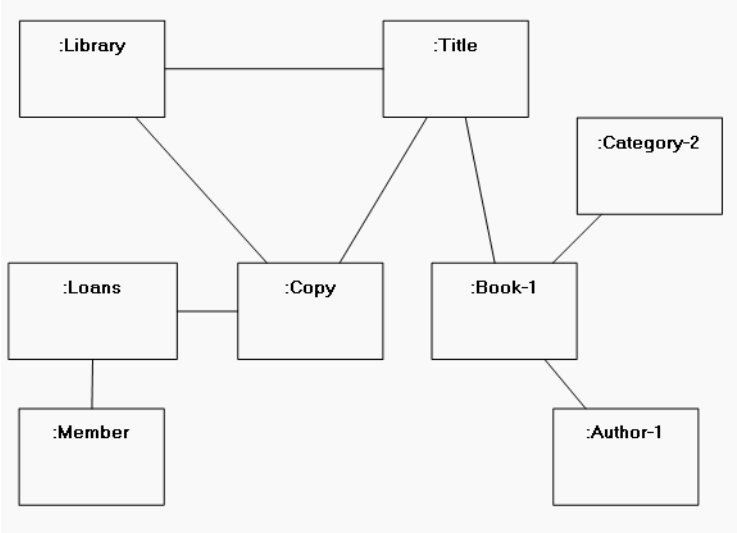
2. Then click one of the associations concerned by the constraint, and drag the mouse to the second association before releasing the mouse button. The Add Constraint dialog box appears.
3. Enter the name of the constraint, then click **Create**.
   The constraint then appears in the drawing.

   ☛ *You can link a constraint to other classes or associations using the **Link** button*

Object Diagram

An object diagram or instance diagram contains objects with values illustrating their attributes and links. It shows in detail the state of the system at a given moment.

You can create the object diagram of a class, component, package or use case.



Objects

An object is an entity with a well-defined boundary and identity that encapsulates state and behavior. Its state is represented by the values of its attributes and its relationships with other objects. Its behavior is represented by its operations and methods. An object is an instance of a class.

Examples of objects:

- Business objects:
  - John Williams, Elizabeth Davis, Paul Smith are instances of the person class.
  - Orders 10533 and 7322 are instances of the order class.
  - Sony SPD-1730 Monitor, Compaq Deskpro 200 are instances of the item class.
  - Dupont and Burger King are instances of the company class.
- Technical objects used for programming:
  - Dlg_Order_Create, Dlg_Client_Query are instances of the window class.
  - Str_Client_Name, Str_Product_Comment are instances of the string class.

☛ *The objects represented in an object diagram can be instances of a class, package, use case, component, or node, to enable defining sequence diagrams at the desired level of detail.*

Creating an object (instance)

To create an object:

1. Click the **Instance** button
   You can create objects of different types. The arrow at the right of the button offers a shortcut to Class and Component object types, the most frequently used.
2. Then click in the diagram workarea.
   The dialog box enabling addition of an instance opens.
3. Enter the instance **Name**.
4. Specify the **Instance Type** if necessary.
5. Click **Create**.

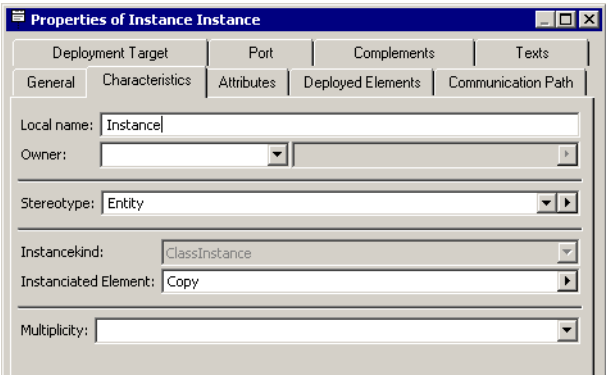
The instance appears in the diagram.

Instance properties

To open the properties dialog box of an instance:

❯ Right-click the instance and select **Properties**.

Several tabs allow you to define the properties of an instance.



You can:

- Select the **Instance kind** (Actor, Class, etc.).
- You can specify of which **Class**, **Actor**, etc. this object is an instance.
- Indicate a name for this instance.
- Specify its **Stereotype**.

Value of an attribute

To specify the value of an attribute:

1. Right-click the class instance containing the attribute.
2. Select **Attributes**.
3. In the dialog box that opens, indicate the value of the attribute. You can specify an instanced value or a constant value.
   - **Instanced value**: click in this column to display the list of possible instances for the selected attribute. These are variable values.
   - **Value**: click in the column and enter the value of the attribute.

Links

A link represents an instance of an association between two objects.

Examples of links between objects:

- Order no. 10733 was placed by John Williams.
- Order no. 10733 includes the products Sony SPD-1730 Monitor and Compaq Deskpro 200.
- John Williams works for Dupont.
- The window Dlg_Client_Query displays the string Str_Client_Name.

Creating a link

To create a link:

1. Click the **Link** button in the toolbar.
2. Click one of the objects concerned, and drag the mouse to the second object before releasing the mouse button.

The link then appears in the diagram.

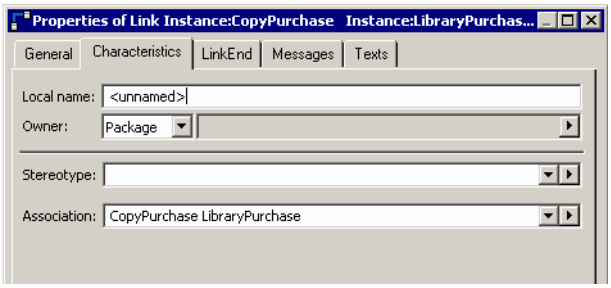
If there is already a link between the two objects, a dialog box asks you to choose an existing link or create a new one.

Link properties

To open the properties dialog box of a link:

- Right-click the center of the link and select **Properties**.

Its dialog box appears.



☛ *If you do not click on the center of the link, the properties dialog box for one of the roles will be displayed.*

In the **Characteristics** tab, you can specify:

- The **Name** of the link.
- The link **Stereotype**.
- The **Association** corresponding to the link.
- The **Package** containing the link.

In the **Link Role** subtab of the link you can specify:

- For each **Instance** connected by this link, the name of the **Role** and its **Multiplicity**.

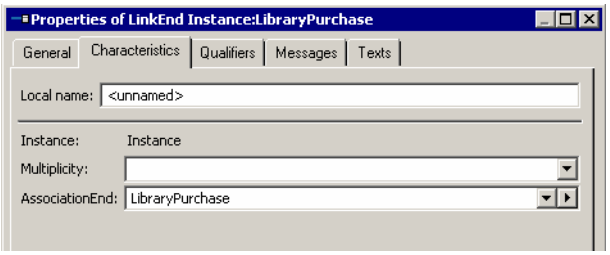
☛ *Only the associations between the classes of the two instances are listed.*

Role properties

To open the properties dialog box of a role:

1. In the Properties dialog box of the link, select the **Link End** tab.
2. Right-click the role and select **Properties**.

The Properties dialog box of the role opens.



☛ *If you do not click the end of the link, the properties dialog box for the link will be displayed.*

In this dialog box you can specify:

- A **Name** for the role.
- The **Role** for this instance.
- The **Multiplicity** for the role.
- The values for the role *Qualifiers*, defined at the class level.

THE CLASS EDITOR

The class editor allows you to display the properties of a package or class and their links in the repository.

✓ "Opening the Class Editor", page 82
✓ "Class Editor Display Parameters", page 82
✓ "Object Properties", page 83
✓ "Creating objects in the class editor", page 84

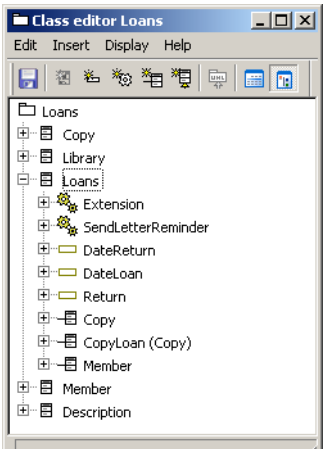
Opening the Class Editor

The class editor is accessed, either in the navigator or in a diagram, from a class or package.

To open the class editor:

1. Right-click a class or package.
2. Select **Class Editor**.

The class editor is displayed. It presents, in the form of a menu tree, the package or class concerned together with its associated objects.



Class Editor Display Parameters

To customize display of the class editor:

- Click **View Toolbars**.
- In the dialog box that opens, select **Filterbar**.

Buttons appear in the toolbar allowing you to display:

Attributes

Operations

Supported interfaces

Inherited classes

Inheritance classes

To keep the editor at the front:

- Click the button.

To refresh the display:

- Click the button.

☛ *These buttons can be filtered. In this case, click **View Toolbars...**.*

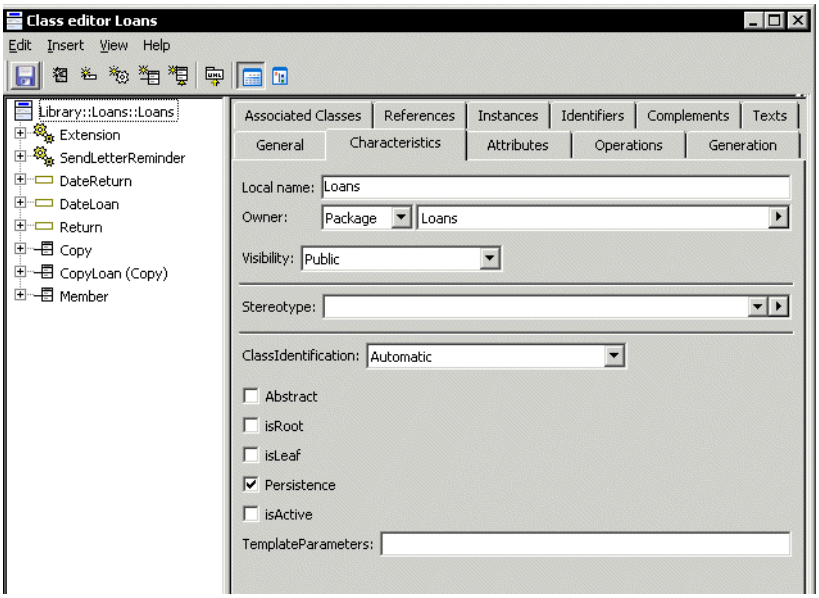
Object Properties

You can display details of objects presented in the menu tree.

To view the details of objects;

- Click the **Details** button

The properties of the selected object are displayed on the right.



As in the properties dialog box, various tabs allow you to check and modify the properties of the object.

☛ *See "Class Properties", page 33.*

Creating objects in the class editor

The class editor allows you to create attributes, operations, associations and inheritances.

For example, to create an operation:

1. Click the class concerned in the menu tree.
2. Then click the **Create operation** button in the editor toolbar.
   The **Create Operation** dialog box opens.
3. Enter the name of the operation and click **OK**.

The operation created appears in the menu tree and in the **Operations** tab of the class properties dialog box.

Creating an association between two classes

To build an association between two classes:

1. In the class editor, select the class for which you wish to create an association.

   ☛ *You can also create an association form from a role.*

2. Click the **Create association** button in the toolbar:
3. In the dialog box that appears, enter the name of the associated class. You can query an existing class or create a new class.
4. Click **OK**.

The association appears in the editor.

Creating a generalization between two classes

To create a generalization between two classes:

1. In the class editor, select the class for which you wish to create a generalization.
2. Click the **Create Inheritance** button in the toolbar.
3. In the dialog box that appears, enter the name of the class that you wish to inherit. You can query this class by clicking the arrow at the right of the inherited class text box.
4. Click **OK**.

The superclass appears in the editor.

GENERATING A CLASS DIAGRAM

You can generate a class diagram from:

- A package
- A class

From a package

- Right-click a package.
- Select **New > Generate a class diagram**.

This command builds a descriptive diagram of the package. It contains all the classes contained in the package, and for each class:

- An inheritance level;
- The first level of association with any association classes.
- The first level of implemented interfaces.

From a class

You can generate two types of class diagram:

- **A simple class diagram**: describes the first level of classes associated with any association classes, the first level of inheritance and the first level of implemented interfaces
- **Complex class diagram**: describes ***n*** levels of classes associated with any association classes (recursion stops if a class C does not belong to the same package as the described class but if c is included in the diagram), the first level of inheritance and the first level of implemented interfaces.

Automatic reorganization

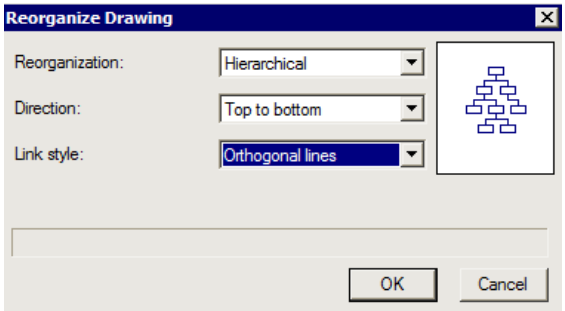
The authorization diagram, class diagram, collaboration diagram, components diagram and the relational diagram all have the automatic drawing reorganization functionality.

☺ *The automatic drawing reorganization functionality is automatically enabled on loading a diagram that does not yet include a drawing.*

To modify organization of an existing drawing:

1. Select **Drawing > Reorganize Drawing**.

2. Select the desired reorganization mode, the direction and the style of links in the diagram.



☺ *The miniature image alongside the reorganization options gives you a view of each type of reorganization.*

3. Click **OK** to apply the modifications.

Structure and Deployment Diagrams

In addition to class and object diagrams, structural diagrams include:

- ✓ The Package Diagram, enabling organization of elements of the model
- ✓ The Component Diagram, highlighting dependency relationships between components
- ✓ Composite Structure Diagram, describing interactions between components and their parts

THE PACKAGE DIAGRAM

A package diagram enables organization of modeling elements, in order to partition the work involved in specification and development.

An element should only appear in a single package.

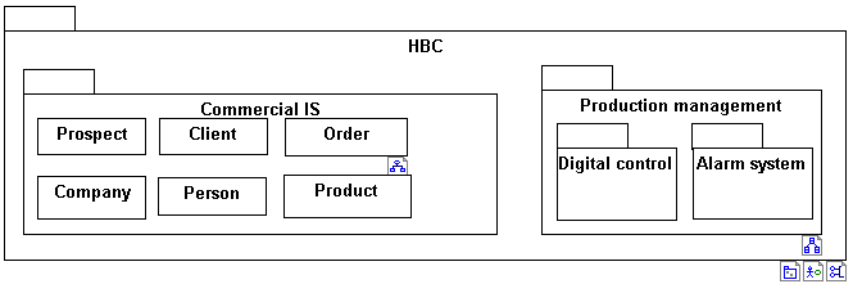
Dividing into packages is generally carried out so as to minimize interactions between different packages.

Example of a package diagram

The "HBC" package contains the "Commercial IS" and "Production Management" packages.

The "Production Management" package can be divided into two packages, "Digital Control" and "Alarm System".

The "Commercial IS" package contains the "Prospect", "Client", "Company", "Person", "Order", and "Product" classes.



Creating a Package Diagram

A package diagram is created from a package. For package creation, see "Creating a Use Case Diagram", page 14.

To create a package diagram:

1. In the **Main Objects** navigation window, right-click a package name.
2. In the pop-up menu that appears, select **New** > **Diagram**.
   A dialog box opens listing the possible diagram types.
3. Select "Package Diagram" and click **Create**.

Defining Packages

A package partitions the domain and the associated work. It is used to organize elements into groups, particularly use cases and classes. Packages may be nested

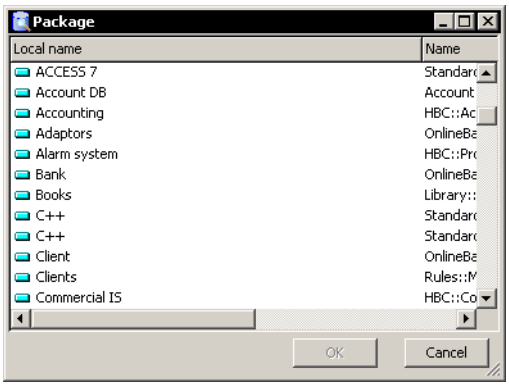
within other packages. Packages are linked by contractual relationships that define their interface.

Examples of *packages*:

- The commercial information system.
- Accounting.
- Production management.
- Digital control of a machine.
- Inventory management.
- Alarm system and telephone management.

To add in the diagram an existing package:

1. Int he package diagram, click the **Package** button in the insert toolbar, then in the workspace.
2. In the **Add Package** dialog box, select **List** in the drop-down list box using the arrow ▶.
   The list of packages appears:



☺ *Note that you can also bring up the list of existing classes by pressing the <Ctrl-L> key.*

You can also enter the first few letters of the name, for example "Pack", and then press <Ctrl-L> to view the list of packages whose names begin with "Pack".

3. Select the desired package and click **OK**.
   The name of the diagram appears in the **Add UML Package** dialog box.
4. Click **Add**.

The package appears in the diagram.

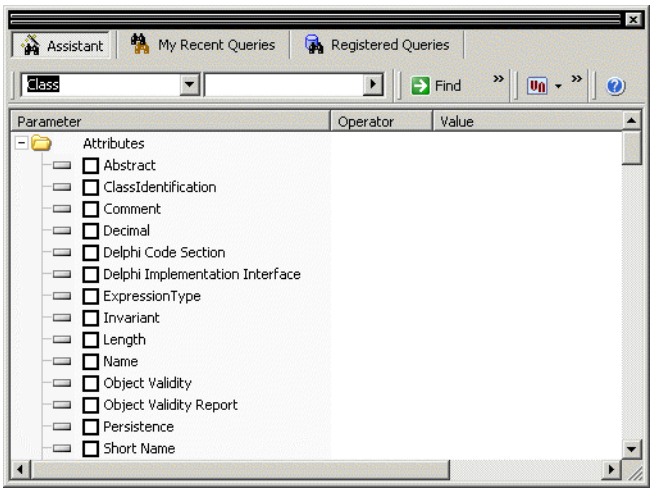
Defining Classes

The package diagram can be used to place classes in different packages.

To add classes to the package diagram:

1. Search for the classes already created by clicking the query button of **MEGA** bar.

2. In the dialog box that opens, select the target "Class".



3. Click **Find**.
   The list of classes opens:
4. Select the classes you want and drop them in the diagram.

Specifying Dependencies in a Package Diagram

Links allow you to indicate if a package contains or references a class or another package.

To indicate that a package references a class or another package:

1. Click the button.
2. Then create the link from a package to the package or class that it references.
   A dialog box asks you the type of link to be created.
3. Select "Referenced package" or "Referenced class" as required.

THE COMPONENT DIAGRAM

A *component diagram* shows the interdependency of software components and *interfaces* (it defines who uses what).

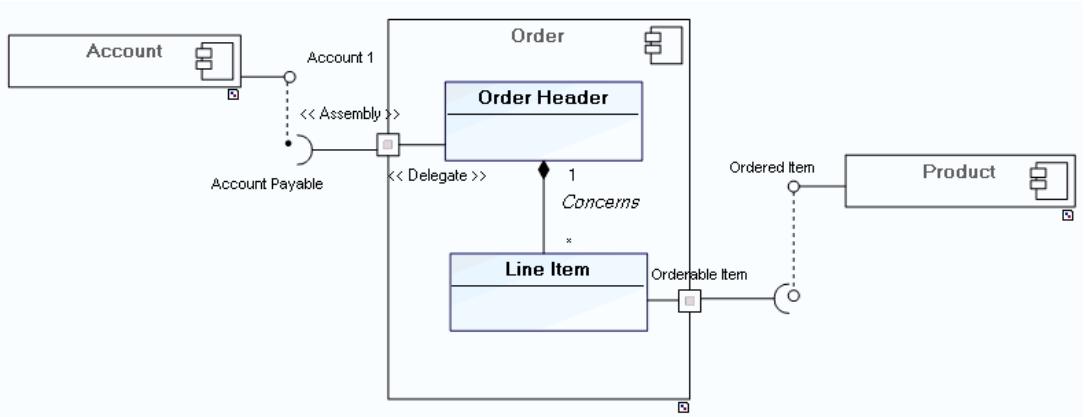
> *A component represents a modular part of a system that encapsulates its content, and which can be replaced in its environment. A component defines its behavior by means of interfaces that it provides and requires.*
>
> *An interface represents the visible part of a class or package in a contractual client-supplier type relationship. The interface is a class stereotype.*

A component diagram contains components and classes of the "Interface" stereotype. It is also possible to specify packages implemented by the components.

You can create a component diagram from a component or package.

Example of a component diagram

This diagram describes the elements contained in the "Order" component and the interactions of these elements with external components.



Components

A component represents a modular part of a system that encapsulates its content, and which can be replaced in its environment. A component defines its behavior by means of interfaces that it provides and requires.

One component can be replaced by another if their interfaces conform.

A component can be a software package, program, code unit, etc.

It is represented by the following icon:

Interfaces

Creating component interfaces

An interface represents the visible part of a class or package in a contractual client-supplier type relationship.

The interface is a particular type of class.

To create a class of "Interface" stereotype in the composite structure diagram:

1. Click the **Interface** button , then click in the diagram.
2. In the dialog box that appears, enter the name of the class.
3. Click **Create**.

☛ *You can specify the details for the interface in terms of attributes and operations in the class diagram in the same way as for a class.*

Linking interfaces to other objects

Two link types enable differentiation of required interfaces and provided interfaces.

A required interface is an interface necessary for object operation.

```
Example: the "Purchasing Management" component requires the
"Product" interface for its operation to be able to
associate a purchase order with products ordered.
```

A provided interface is an interface made available by an object to other objects.

```
Example: the "Product Management" component makes available
the "Product" interface.
```

You can define interfaces required and provided by an object independently of other objects.

To specify that an interface is supported by an object:

1. Click the button.
2. Draw the link between the provider object (component, package or class) to the supported interface.

To indicate that an object requires an interface:

1. Click the button.
2. Then create the link from the client object to the required interface.

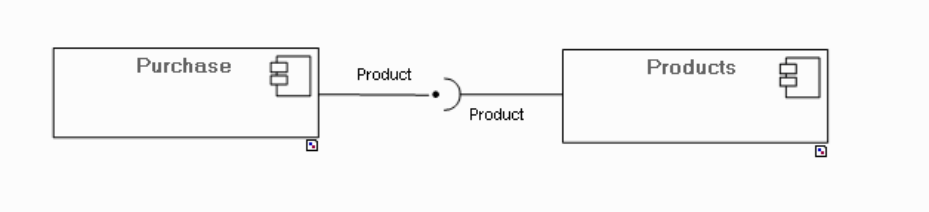
☛ *You can also specify dependencies between packages or between packages and classes using the button as in the package diagram.*

According to the link type, a required interface is represented by a half-circle, a provided interface by a circle.

Connecting interfaces

Two interfaces can be interconnected. This connection is modeled by a connector.

You can also indicate that an interface provided by an object is required by another. Here it is one and the same interface.



Ports

Ports enable connection of a component to its parts or to its environment.

Ports are represented by a square in the diagram, placed at the edge of the described element when they assure connection with the exterior.

They are connected to components by connectors.

Ports can specify queries sent and services provided by the component, as well as queries and services they may require from other parts of the system. These queries and services are represented by classes of interface type.

You can view interfaces associated with a port in the properties dialog box of the port, in the **Provided and Required Interfaces** tab.

Connectors

Connectors enable connection of diagram objects.

Connectors of simple type do not specify a particular connection type, they are notably used to connect instances of objects described in collaborations.

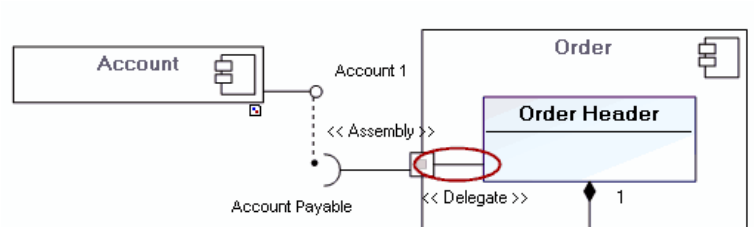
In the composite structure diagram, it is possible to specify the type of connector between two objects: Assembly or Delegate.

Delegate connector

A "Delegate" type connector indicates the redirection of queries to a component element responsible for their execution.

The delegation link can be made directly between the component port and the component element, or between the component port and the element port.

Below, the "Order" component delegates management of accounts to be debited to the "Order Header" class.



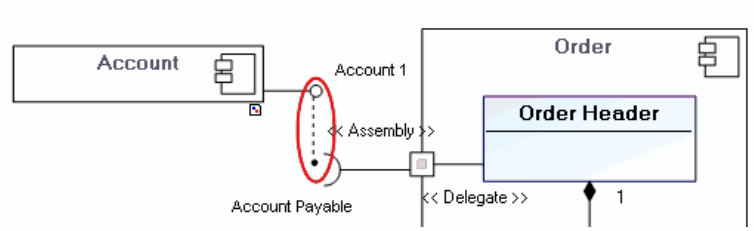
Assembly connector

An "Assembly" type connector is a connector between two or more components or ports indicating that one or more components provide services that others use.

☛ *These can be other objects or components.*

To connect ports or components that share an interface, you can also use "Provided Interface" and "Required Interface" links.

An "Assembly" type connector connects the interface provided by the "Account" component to the interface required by the "Order Header" class.



Composite Structure Diagram

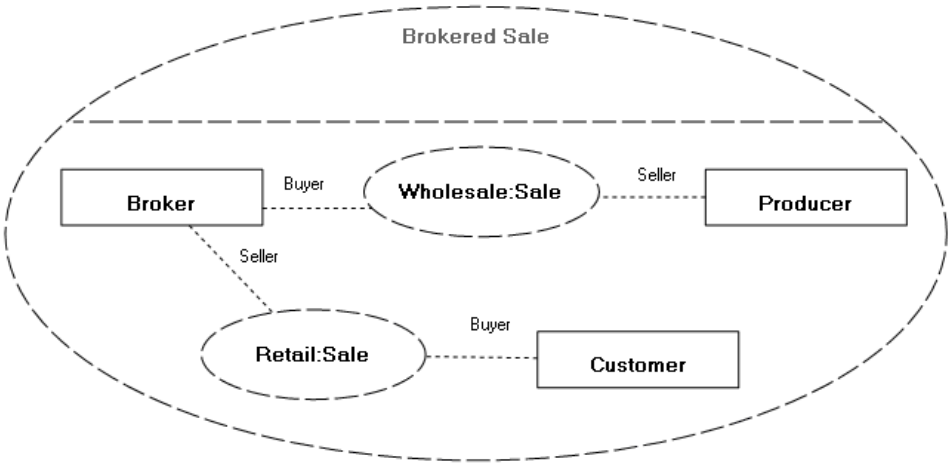
The composite structure diagram enables description of the internal structure of a component, a package or a structured class.

It also enables specification of collaborations that intervene between elements of the structure in execution of a task, highlighting the role played by each element in the collaborations.

Elements of this diagram are *parts*, ports by which parts interact with the exterior, and connectors linking the parts between themselves and with the ports.

Example of a composite structure diagram

This diagram describes the role played by parts in the "Brokered Sale" collaboration.



Parts

A part represents a role played by an instance of a class or component at execution of a task.

Parts are interconnected by connectors or dependencies.

A part can also be connected, via a connector, to a port which acts as interface between the described component and the exterior.

For more details on these elements, see:

✓ "Connectors", page 95
✓ "Dependency links", page 99
✓ "Ports", page 95.

Multiplicities defined on parts indicate the number of instances created. Multiplicities on connector roles indicate the number of links that can be created for each of these instances.

To define multiplicity of a part:

1. Open the properties dialog box of the part.
2. Click the **Characteristics** tab.
3. Click the arrow in the **Multiplicity** box and select the required multiplicity.
4. Click **OK**.

Collaborations

In the composite structure diagram, a *collaboration* describes the role played by each part (instance) in execution of a task.

> *A collaboration (UML) describes a collaborative structure between several elements (roles), each accomplishing a specialized function and collectively producing an expected functionality of the system. Its objective is to show how a system functions independently of a specific use. We therefore generally remove the precise identity of the participating classes or instances.*

It is represented by a dotted line oval containing the collaboration instances.

These instances are interconnected by *connectors*. The role that corresponds to the instance name is displayed at each end of the connector.

> *A connector is a link that enables communication between several objects. A delegation connector connects the external contract of the object (as specified by its ports and/or interfaces) to internal objects that will produce it. An assembly connector between several objects (or their ports) enables specification of how one of the objects provides the interface required by another.*

The model of a collaboration can be applied to different instances.

Collaboration use

A collaboration use represents application of the structure described by a collaboration to a particular situation implementing classes or specific instances. These classes or instances therefore play roles defined in the collaboration.

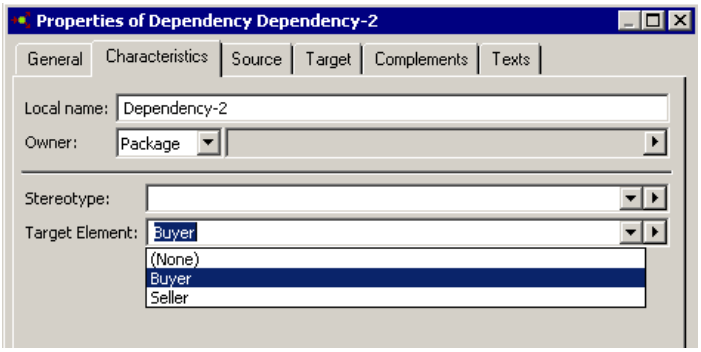
The instances are connected to the collaboration use by a *dependency* link on which the role played by the instance must be specified.

> *A dependency specifies that implementation or operation of one or several elements requires the presence of one or several other elements. There are several dependency stereotypes.*

Collaboration use example

```
In the case of a purchasing request between two instances of
an actor, a collaboration is used. This collaboration
connects two roles: the role of buyer and the role of
seller. On the dependency that connects each instance to the
```

```
collaboration, you can indicate the role played by the
instance.
```



Dependency links

A dependency specifies that implementation or operation of one or several elements requires the presence of one or several other elements.

A dependency is a supplier/customer type relationship indicating source and target elements in the collaboration.

A stereotype on the dependency enables specification of dependency type:

- Binding: relationship between a template and a modeling element generated from the template. It includes a list of arguments corresponding with template parameters.
- Derive : indicates a derivation relationship between modeling elements that are generally, but not necessarily, of the same type. Such a dependency relationship implies that one of the elements can be calculated from the other.
- UML/XML Mapping : a mapping expression that defines the relationship between elements (classes, attributes, etc.) of a schema or class diagram and those of another schema or class diagram.
- Refine: specifies a dependency relationship between modeling elements at different semantic levels, such as analysis and design.
- Trace: specifies a traceability relationship between modeling elements or sets of modeling elements that represent the same concept in different models.

To specify dependency type:

1. Open the properties dialog box of the dependency.
2. Click the **Characteristics** tab.
3. In the **Stereotype** box drop-down list, select one of the proposed stereotypes.

   The arrow ▶ also allows you to create new stereotypes.

STATE MACHINE DIAGRAM

A state machine diagram enables description of possible behaviors of an object, depending on the events it experiences during its life cycle.

The following points are covered here:

- ✓ "Presentation of the State Machine Diagram", page 102
- ✓ "Creating a State Machine Diagram", page 102
- ✓ "States", page 103
- ✓ "State Transitions", page 107

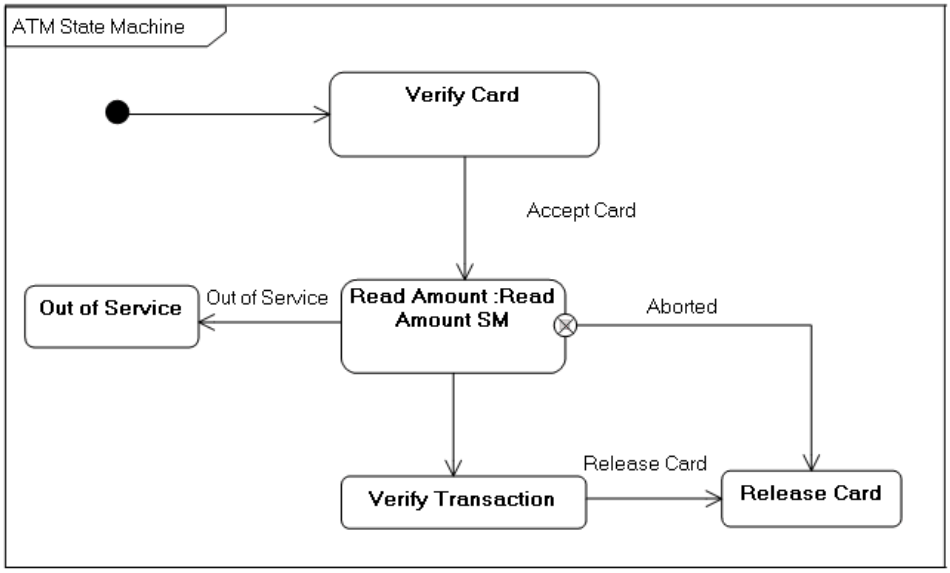
Presentation of the State Machine Diagram

A state machine is the set of states and transitions between states that define the life cycle of an object that is variable over time.

The state machine diagram enables representation of the sequence of states that an object can take in response to interactions with the objects (internal or external to the studied system) in its environment.

Example of state machine diagram

The diagram below describes possible behaviors of an automated teller machine:



Creating a State Machine Diagram

A state machine diagram is created from a state machine.

You can create a state machine from a package, class or component.

To create a state machine diagram:

1. Right-click the state machine.
2. Select **New > State Machine Diagram**.

The diagram is initialized by creation of a region. A region is part of a composite state or state machine which contains states and transitions and of which execution is autonomous.

STATES

A state is a condition or situation in the life of an object, during which it satisfies some condition, performs some activity, or waits for some event. A state represents an interval of time delimited by two events. It is a phase an object passes through during its life cycle.

Examples of object states

- A person can be:
  - Unmarried
  - Married
  - Divorced
- An item can be:
  - Available
  - In stock
  - At reorder level
  - Out of stock
  - etc.

Creating a State

To create a state:

1. Click the arrow associated with the **State** button of the object insert toolbar
2. Select a state type.
3. Click in the diagram work area.
   The **Add State** dialog box opens.
4. Indicate the **Name** of the state and click **Create**.
   The state appears in the diagram.

State types

It is necessary to specify the state type at the time of its creation. It can be:

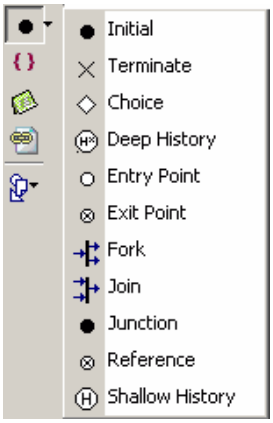
- A normal state: has no sub-structure.
- A composite state: comprises several states, described in the diagram.
- A sub-machine state: calls the descriptor of a state machine described elsewhere. See "Detailing Behavior of a State", page 105.
- A final state

When you place a state in another state, it is automatically connected as a component of this state.

Pseudo-states

Pseudo-states are used to specify complex paths by combining several transitions between states.

They can be of different types: initial, final, choice, deep history, shallow history, input, output, fork, join, junction or reference.



Initial

An initial pseudo-state has a single output transition to the initial state of the object at its creation.

Deep history

A deep history pseudo-state represents the last active configuration of a composite state containing it; that is the configuration that was active the last time the composite state was exited.

Simple history

A simple history pseudo-state represents the most recent active sub-state of a composite state (without the sub-states of this sub-state).

Fork

A fork separates a transition into several concurrent transitions.

Join

A join is the grouping of several transitions into a single transition.

Choice

Represents the choice of a transition between several possible transitions.

Junction

A junction is used to define paths of complex transitions between several states.

Input

Entry point of a state machine or of a composite state.

Output

Exit point of a state machine or of a composite state.

Reference

Reference to an input or output of a state machine or of a composite state.

Final

Input in this pseudo-state involves complete shutdown of the state machine.

Deep history

A **Deep History** state represents the last active configuration of a composite state; that is the configuration that was active the last time the composite state was exited.

A **Simple History** state represents the most recent active sub-state of the composite state.

Example:

Consider the "Married" state as the last active configuration. Sub-states of this state are "With children" and "Without children". In the case of a deep history, the "With children" and "Without children" sub-state is specified. In the case of a simple history, only the "Married" state is taken into account.

Detailing Behavior of a State

A state can be made up of sub-states.

To describe composition of a state in a diagram:

1. Open the pop-up menu of a state and select **New** > **Detailing Behavior**.
   The state machine diagram creation window opens.
2. Click **Create**.
   The diagram opens.

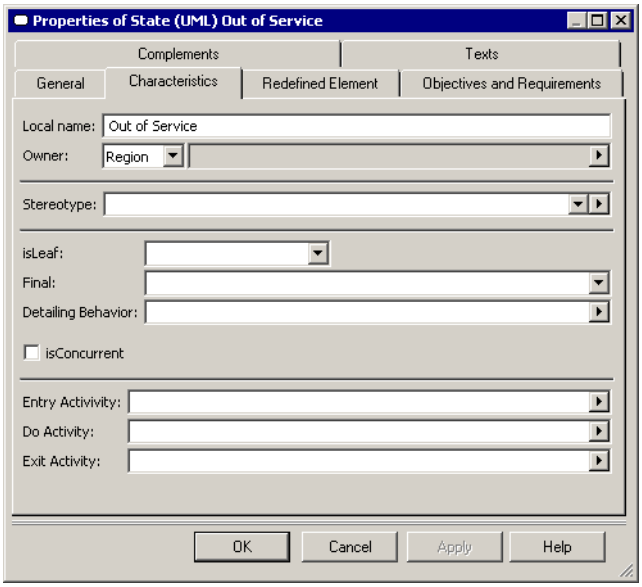
You can also define composition of a state by associating it with a new or existing state machine:

1. Open the properties dialog box of the described state.
2. Click the **Characteristics** tab.
3. In the **Detailing Behavior** box, create a state machine or query an existing state machine.

State Properties

To access the state properties:

1. Right-click the state.
2. Select **Properties**.
   The properties dialog box of the state appears:



This can be used to:

- Modify the state **Name**.
- Indicate whether the sub-states are **Concurrent**, meaning they can be executed simultaneously.
- Indicate the **Detailing Behavior** (in the case of a complex state). See "Detailing Behavior of a State", page 105.
- Specify the **Activities** that can be performed at input, output or while the object is in this state.

☛ *The contents of the properties dialog box of a state vary depending on state type.*

STATE TRANSITIONS

Passage from one node to another is represented by a *transition*.

A transition indicates the change of an object from one state to another. A transition is a response of an object to an event it receives. When a specified event occurs and certain conditions are satisfied, the object will perform certain specified actions while it is still in the first state and then enter the second state.

All authorized transitions must be defined. Those that are not defined are prohibited.

Examples of transitions:

For the marital status of a person, certain transitions are possible:

- It can change from the "unmarried" to the "married" state
- It can change from the "married" to the "divorced" state.

Other transitions are not possible:

- The state cannot change from "unmarried" to "divorced".

Creating a Transition

To create an transition between two states:

1. Click the **Transition (UML)** button in the object insert toolbar.
2. Click the source state and drag the mouse to the target state.
3. Release the mouse button. The association is created.

Transition Types

A transition can be external, internal or local.

You can specify the transition type in the transition properties dialog box, in the **Characteristics** tab.

External transition

An external transition is a transition that modifies the active state.

Internal transition

An internal transition enables an object to react to the arrival of an event that does not result in a state change but has an effect such as calling an operation or sending a message. For example, when pulling items from inventory, an item may not change state if the quantity remaining in the inventory is sufficient and does not fall below the reorder level or shortage level.

Local transition

A local transition applies to sub-states of a composite state. It can cause a change of state only within the composite state.

Transition Effects

Triggering of a transition can be accompanied by an effect. The effect can be represented by:

- An activity
- A collaboration
- An interaction
- A state machine

To define effect of a transition:

1. Open the properties dialog box of the transition.
2. Select the **Characteristics** tab.
3. Click the arrow in the **Effect (Behavior)** box and create or connect the object that defines the effect.

Transition Effect Display

To modify how the transition effects are displayed.

1. In the state machine diagram, right-click the transition and select **Shapes and Details**.
2. Then select "Effect" in the tree that appears.

You can now specify whether to display all or part of the transition effects and their characteristics.

Transition Triggering Event

In the properties dialog box of a transition in the **Event** tab, you can indicate the **Event Kind** that triggers a transition.

It can be:

- Any event
- Calling an operation
- Changing the object concerned by the transition
- Creating an object
- Destruction of an object
- Sending a signal
- Sending an operation
- Sending a signal from the object
- Receiving a signal
- Receiving an operation
- A *time mark*

> *A timer is an event determined only by the passage of time. Example: Monday, 4:00 pm, etc.*

Fields displayed under **Event Kind** vary according to the event kind selected.

You can select the object concerned by the effect.

In the case of an operation or signal, you can specify values of parameters sent. vérifier + capture

ACTIVITY DIAGRAM

The activity diagram is very similar to the state machine diagram. Unlike the state machine diagram which describes object behavior via state sequencing, the activity diagram describes element behavior in terms of actions.

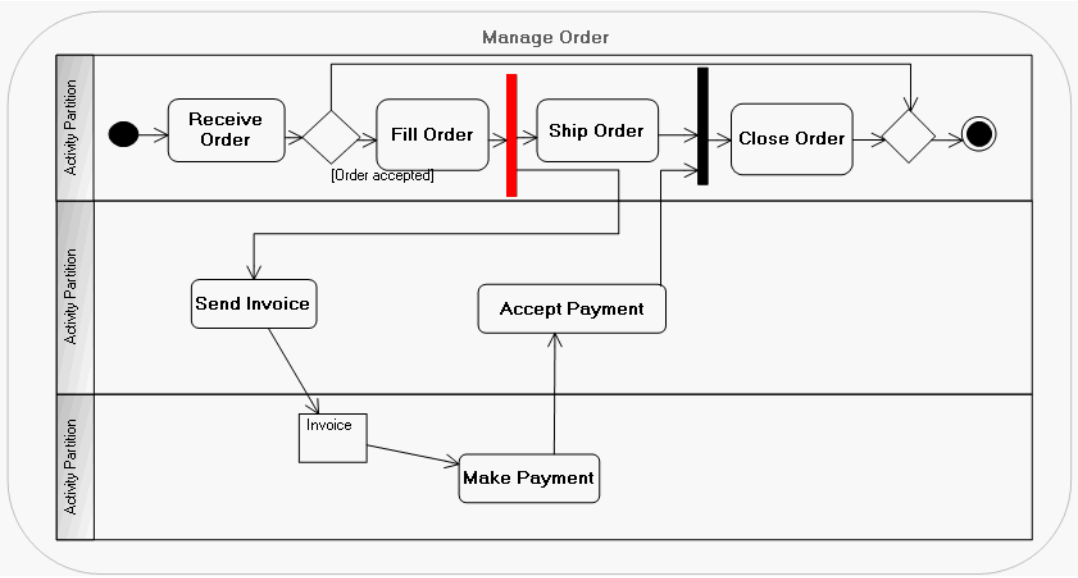
- ✓ "Activity Diagram", page 112
- ✓ "Partitions", page 113
- ✓ "Nodes", page 115
- ✓ "Flows", page 117

Activity Diagram

An activity diagram represents sequencing of steps describing behavior of a system element.

Steps are modeled by nodes - nodes of action, configuration or control - coordinated by data flows or control flows.

Example of an activity diagram



Creating an Activity Diagram

An activity diagram is created from an activity.

You can create an activity from a package, component or class.

To create an activity diagram:

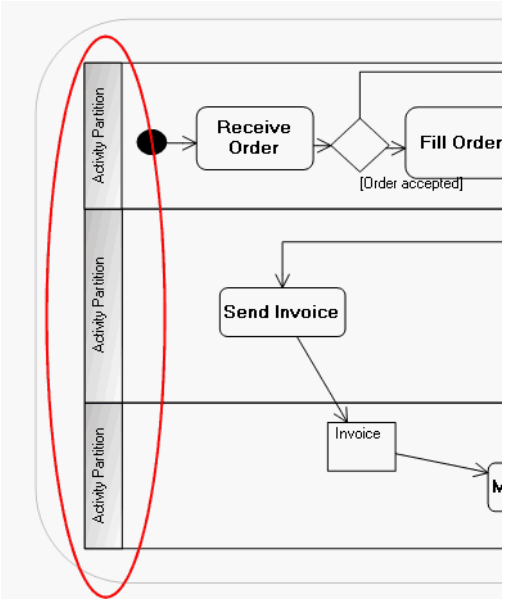
1. Right-click the activity concerned.
2. In the pop-up menu that appears, select **New** > **Diagram**.
3. In the dialog box that opens, select "Activity Diagram" and click the **Create** button.
   The new activity diagram opens.

PARTITIONS

An activity diagram can be divided into partitions. Each partition contains nodes or actions as well as the flows between these elements.

You can use partitions to organize tasks or to specify the element responsible for implementation of multiple tasks.

For more details on swimlanes, see the **HOPEX Common Features** guide, "Handling Repository Objects" chapter, "Using Swimlanes" section.



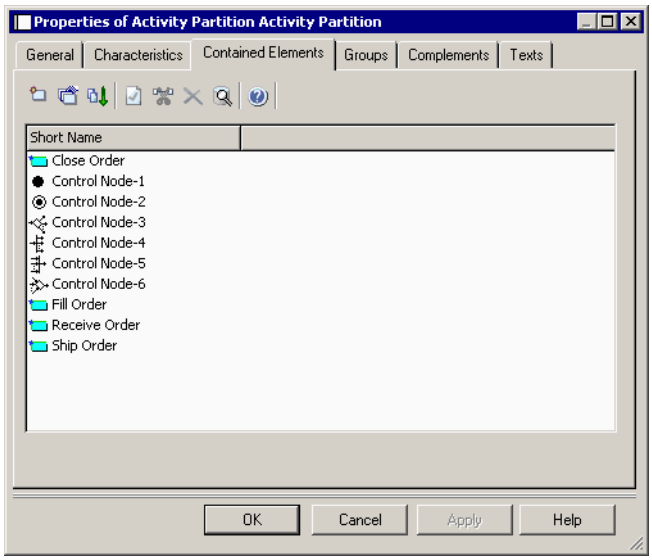
Creating a Partition

To create a partition in the activity diagram:

1. Click the **Partition** button in the object insert toolbar.
2. Specify its name.
3. Indicate the element represented by the partition. This is the element which implements elements of the partition. It can be an actor, class or component.
4. Click **OK**.

Partition Properties

The **Contained Elements** tab of the partition properties dialog box presents elements executed in the partition.



The **Complements** tab enables attachment of the partition to the constraints that are implemented.

NODES

Nodes enable modeling of activity steps. There are different node types in **HOPEX**:

- "Object nodes", page 115
- "Parameter nodes", page 115
- "Control nodes", page 116
- "Object nodes: Input, Output and Exchange Pins", page 117

Object nodes

Actions are the basic steps of behavior represented by the activity.

Coordination of actions is by control flows and data flows.

Creating an Action

To create an action:

1. Click the button in the insert toolbar corresponding to the type of action, then click in the workarea.
   The dialog box for adding an action of the selected type opens.
2. Specify its name and click **Create**.

Action kind

In the properties dialog box of the action, in the **Characteristics** tab, you can specify the action kind. It can be:

- Calling an operation of another object
- Creating an object
- Destruction of an object
- Local execution of an operation of the object
- Sending a signal from the object
- Terminating the object
- etc.

Parameter nodes

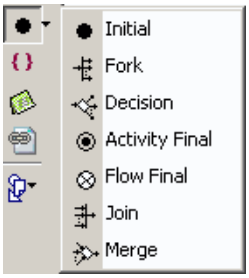
Parameter nodes of an activity describe inputs and outputs of this activity.

They transmit parameters to the activity via flows which they send and receive.

Control nodes

Control nodes coordinate flows between nodes of an activity.

A control node can be of type initial, final, decision, merge, fork or join.



Control node types

Initial

An initial node indicates where the control flow starts when the activity is invoked. An activity can have several initial nodes.

Final

When a token reaches a final node of an activity, all flows of the activity are stopped. Conversely, a final node of a flow destroys tokens that arrive, but has no effect on other tokens of the activity.

Decision

A decision makes a choice of one flow from among several possible output flows. Output flows are selected according to their guard conditions.

Merge

A merge fusion (merge) groups several alternative input flows into a single output flow. It is not used to synchronize concurrent flows, but to accept a single flow from among several.

Fork

A fork separates a flow into several concurrent flows. Tokens arriving at a fork are duplicated through the output flows.

Join

A join synchronizes multiple flows. The flow is triggered when all input flows are available.

Object nodes: Input, Output and Exchange Pins

To specify input values of an action and return values, we use object nodes called input and output pins. The action can only start when a value is assigned to the input pin. Similarly, when the action is completed, a value must be assigned to the output pin.

Input pin

An input pin supports input values consumed by an action that it receives from other actions.

Output pin

An output pin supports output values produced by an action and supplies these values to other actions through flows.

Exchange pin

An exchange pin is used to represent data exchanged between two actions.

Flows

Passage from one node to another is represented by a flow.

Control flow

A control flow starts an action node when the previous node is completed. Objects and data cannot be transmitted by a control flow.

Object flows

An object flow enables transmission of data or objects from one node to another within an activity.

INTERACTION DIAGRAMS

Interaction diagrams, that is the sequence diagram, communication diagram and interaction overview diagram, represent a series of interactions between objects, ordered in time. They show one or more possible illustrations of a system.

The following points are covered here: :

- ✓ "Interactions", page 120
- ✓ "Sequence Diagram", page 121
- ✓ "Communication Diagram", page 133
- ✓ "Interaction Overview Diagram", page 135

INTERACTIONS

An interaction describes behavior of a system in a particular context by exchanges of messages between system elements.

While state machine diagrams or activity diagrams study individual behaviors, interaction diagrams concentrate on cooperation of a group of objects.

Creating an Interaction

You can create an interaction from:

- A component
- A package
- A class

Creating an Interaction Diagram

The sequence diagram, communication diagram and interaction overview diagram are created from an interaction.

To create an interaction diagram:

1. Right click an interaction..
2. In the pop-up menu that appears, select **New** > **Diagram**.
   A dialog box opens listing the possible diagram types.
3. Select the required diagram type and click **Create**.

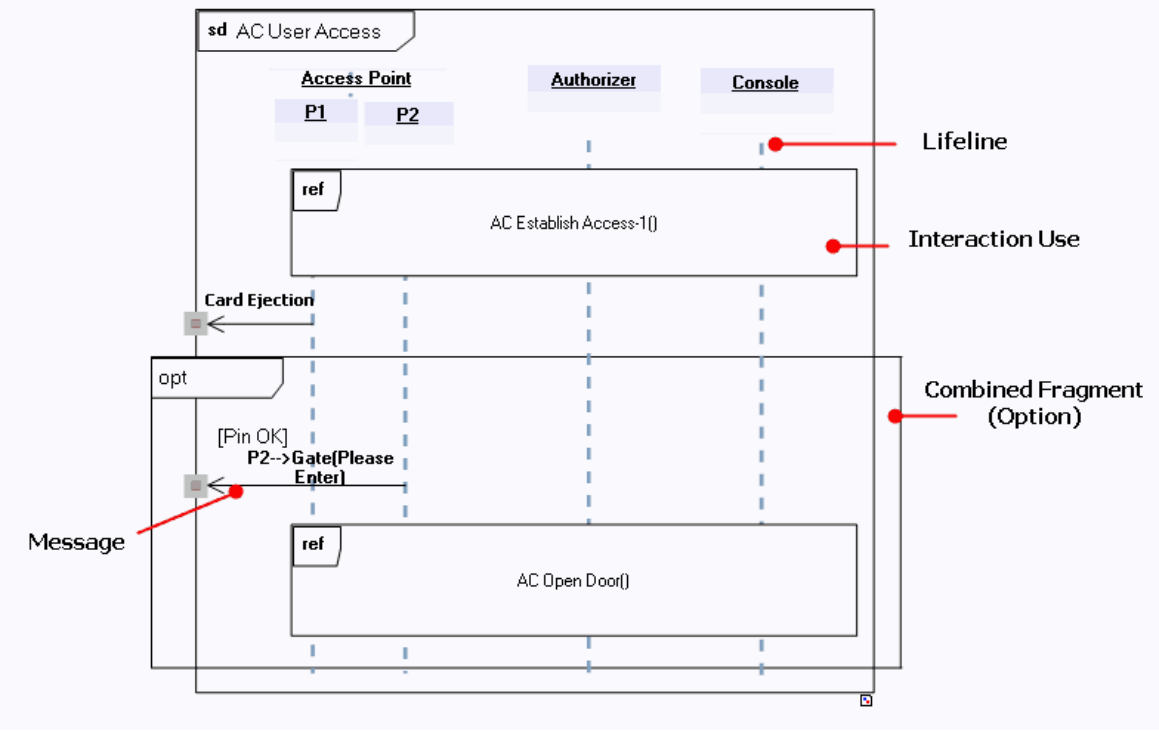
SEQUENCE DIAGRAM

The sequence diagram highlights the chronology of messages exchanged between objects participating in an interaction. These objects are represented in the diagram by their lifelines.

Example of a sequence diagram

The diagram below describes behavior of an automated teller machine:

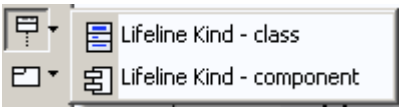
- Two entry points (represented by lifelines) have a user access check. This check is described in an interaction.
- Depending on the result of the check, either access is refused and the user card is rejected, or door opening is actuated;
- An optional behavior (represented by a combined fragment) can influence door opening.



Lifelines

A lifeline represents a participant in an interaction.

Lifelines are instances of different types (of classes, of actors, etc.). The arrow at the right of the button offers a shortcut to Class and Component object types, the most frequently used.



In a sequence diagram, time is represented as passing from top to bottom along the lifelines of these objects. Message instances transit between these objects.

☛ *The instances represented in a sequence diagram can be instances of a class, actor, package, use case, component, or node, used to define the sequence diagrams at the desired level of detail.*

Creating a lifeline

To create a lifeline/

1. Click the **Lifeline** button
2. Click on the diagram.
   A dialog box opens.
3. Enter the name of the lifeline.
4. Indicate the element represented by the lifeline.
5. Click **Create**.
   The lifeline appears in the diagram.

To insert several existing objects into the diagram at once.

1. Click successively on the lifeline button and the Query button
2. In the dialog box that opens, select **Find**.
3. Holding the <CTRL> key down, select the desired objects from the list, then drag-and-drop them in the diagram.

Lifeline properties

To access properties of a lifeline:

- Right-click the instance and select **Properties**.

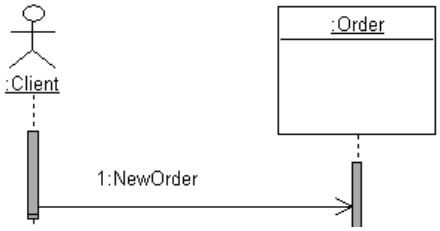
You can select the **Type** of the object (Actor, Class, etc.), specify the **Class**, **Actor**, etc. of which it is an instance, and indicate its *Stereotype*.

Messages

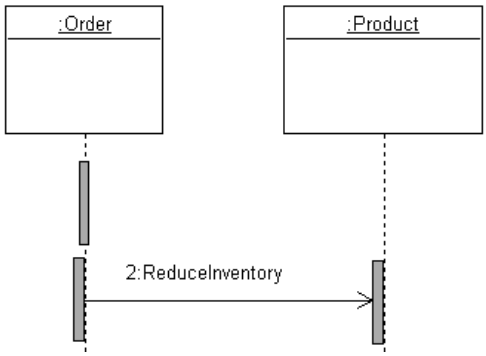
A message defines a particular communication between lifelines of an interaction. It specifies the sender and receiver via intermediate occurrence specifications, as well as the type of communication. This communication can be, for example, sending a signal, calling an operation or deleting an instance.

Examples of exchanged messages

1) The message sent by the "Client" actor to the "Order" class carries the "New Order" signal.



2) The message sent by the "Order" class to the "Product" class calls the "Reduce inventory" operation.



Creating a message

To create a message in the sequence diagram:

1. Click the **Message** button in the insert toolbar, selecting the required message type.

2. Click on the dotted line under the first object, and hold down the mouse button while dragging the cursor to the dotted line under the second object.
   The message exchanged between the two objects is drawn.

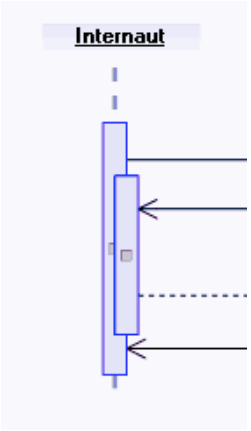
Message types

You can create four types of message:

- In a message type "Complete", the sender and receiver are both defined.
- In a message type "Lost", only the sender is known. Here we consider that the message never reaches its destination.
- In a message type "Found", only the receiver is known. This is the case when origin of the message is outside the description context.
- In a message type "Unknown", neither sender nor receiver are defined.

Execution Specification

An execution specification represents an action or behavior unit that progresses from a start occurrence specification to an end occurrence specification.



Creating an execution specification

To create an execution specification:

1. In the sequence diagram, click the **Execution Specification** button in the object insert toolbar.
2. Position it on the lifeline concerned.
   The specification appears in the diagram.

Occurrence specification

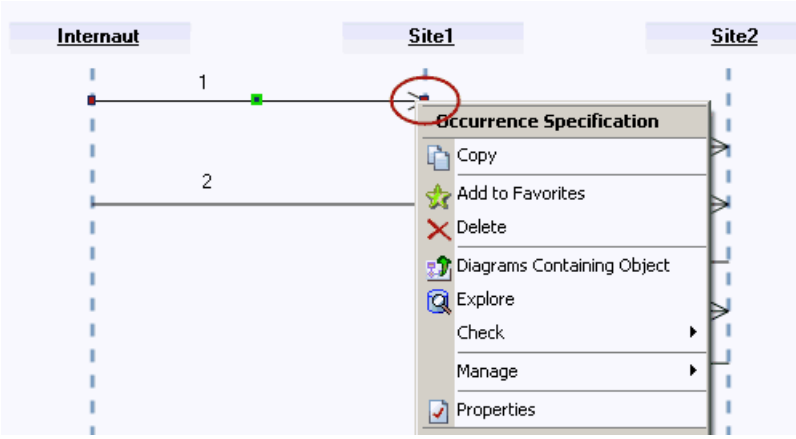
Creation of a message or an execution specification automatically creates occurrence specifications.

An occurrence specification is a syntax point at the extremity of a message or at the start or end of an execution specification.

Occurrence specifications are ordered along a lifeline.

These are basic semantic units of an interaction.

You can access the pop-up menu of an occurrence specification by right-clicking one of the extremities of a message.



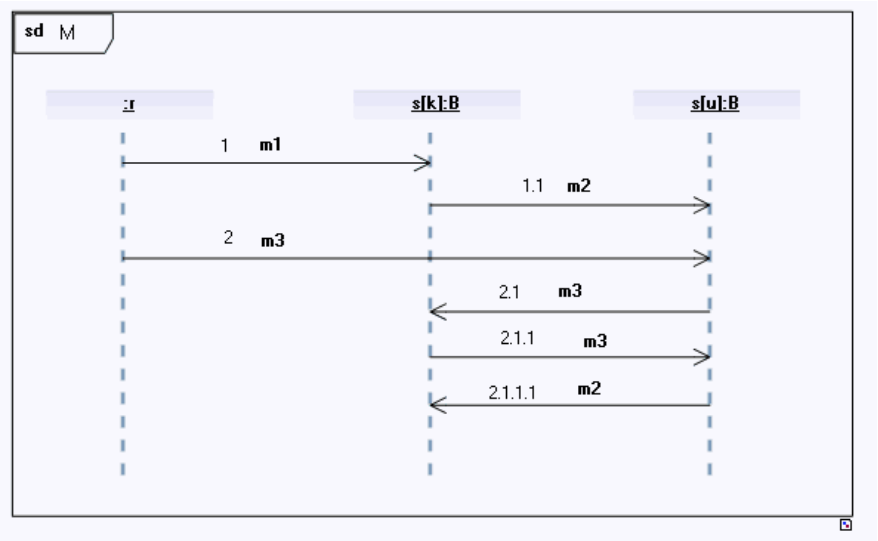
Calculating sequence numbers

From positioning of occurrence specifications, a calculation tool enables ordering of messages and execution specifications.

To order messages circulating between lifelines of an interaction:

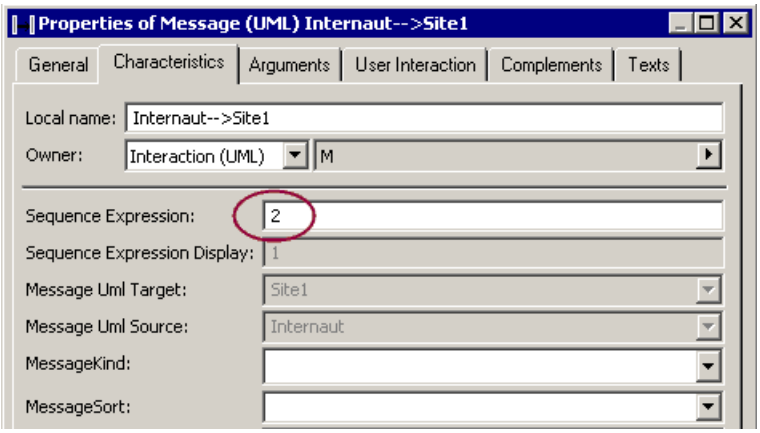
1. Open the pop-up menu of the described interaction.
2. Select **Calculate Sequence Numbers**.
   The tool automatically applies numbers to messages.

Example



You can manually modify the sequence number of a message in the message properties dialog box:

- Select the **Characteristics** tab and change the value in the **Sequence Expression**.



When you restart calculation of sequence numbers, this updates sequencing according to the modifications made.

Combined Fragment

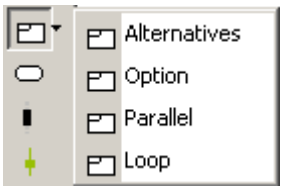
A combined fragment enables concise description of several execution sequences.

A combined fragment is defined by an interaction operator and the corresponding interaction operands.

Creating a combined fragment

To create a combined fragment:

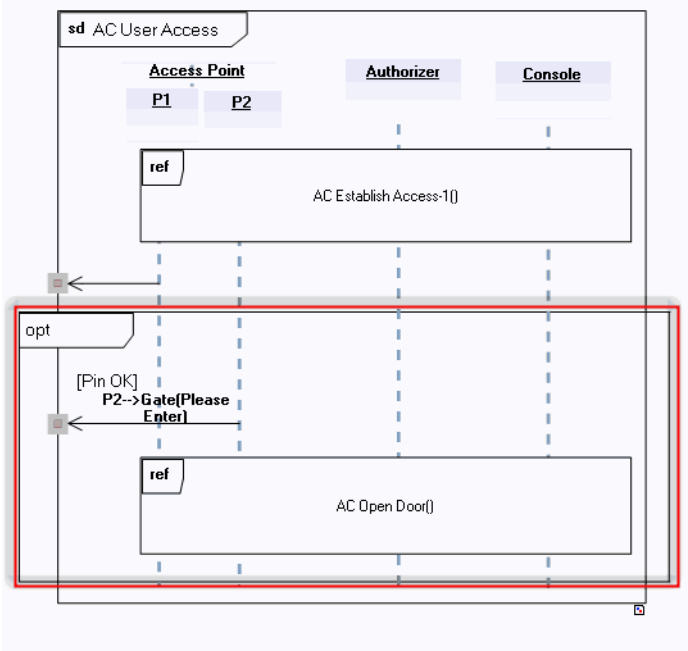
1. In the sequence diagram insert toolbar, click the **Combined Fragment** button.
   You can associate different types of interaction operator to a combined fragment. The arrow at the right of the button offers shortcuts to four of these. See "Interaction operator type", page 128.



2. Click on the diagram.
   The combined fragment creation dialog box appears.
3. Specify its **Name** and the **Interaction Operator Type** if not already indicated.
4. Click **Finish**.

A combined fragment is represented by a rectangle with the interaction operator type displayed at the top left-hand corner.

In the example below, a combined fragment of option type translates a behavior that could disturb normal operation (door opening).



Interaction operator type

The interaction operator type conditions meaning of the combined fragment. There are various operator types: seq, alt, opt, break, par, strict, loop, region, neg, assert, ignore and consider.

Alternatives

Alt expresses the possibility of choosing between different possible behaviors by evaluating guard conditions associated with each of the operands. Only one of these operands can be executed.

The Else operand is selected when none of the other conditions is satisfied.

Option

Opt represents a choice between the unique operand proposed, or none.

Break

Break represents a stop scenario that is executed instead of the rest of the containing interaction fragment.

Parallel

Par means that the different operands can be executed in parallel. Occurrence specifications of different interaction operands can be sequenced in various ways as long as the order imposed by each operand is maintained.

Weak Sequencing

Seq designates weak sequencing between behaviors of operands defined by three properties:

- Order of occurrence specifications within each of the operands is maintained in the result.

- Occurrence specifications of different lifelines from different operands can appear in any order.

- Occurrence specifications of the same lifeline from different operands are ordered so that the occurrence specification of the first operand appears before that of the second.

Strict Sequencing

Strict defines strict sequencing of operand behaviors.

Negative

Neg represents an invalid operand.

Critical Area

Critical represents an area that must be processed atomically, meaning that occurrence specifications cannot be sequenced with those of this critical area.

Ignore/Consider

Ignore and consider require that a list of relevant messages be specified.

Ignore indicates that the types of certain messages are ignored in the combined fragment.

Consider indicates that certain messages will be considered in the combined fragment. This is equivalent to defining all other messages as 'ignored'.

Assertion

Assert represents a sequence that is the only one valid for a given message.

Therefore any sequence defined by an interaction fragment that starts with messages leading to the sequence defined by the Assert block and continuing with an exchange of messages that do not respect the Assert block must be defined as invalid.

Assertions are frequently used in combination with Ignore and Consider types.

Loop

Loop indicates that the interaction operand will be repeated a certain number of times. It is possible to specify minimum and maximum number of loops, as well as an expression of loop continuation.

Interaction operands

An interaction operand is contained in a combined fragment, and represents an operand of the expression given by the containing combined fragment. It can be conditioned by an interaction constraint, which acts as guard condition.

Creating an Interaction Operand

To create an interaction operand:

1. Right-click the combined fragment which contains the interaction operand.
2. Select **New** > **Interaction Operand**.
3. Name the operand and click **OK**.

Creating an Interaction Constraint

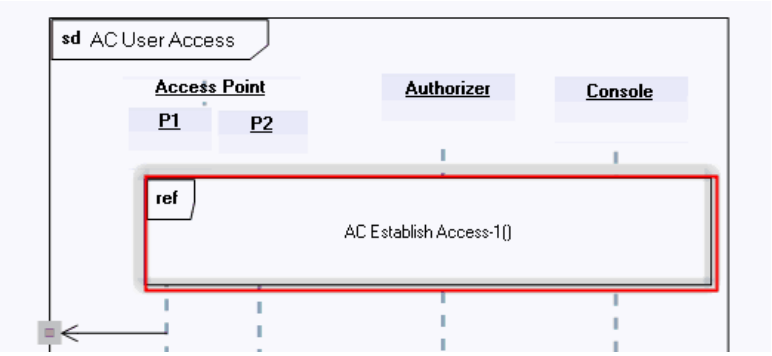
To create the interaction constraint that will condition the operand:

1. Open the properties dialog box of the interaction operand.
2. Click the **Characteristics** tab.
3. In the **Condition** frame, click **New**.
4. The condition is represented by a constraint. Define the constraint and click **OK**.

Interaction Use

An interaction use refers to an interaction. It is a means of copying content of the interaction referenced at the interaction occurrence location.

Example



To create an interaction use:

1. Click the **Interaction Use** button
2. Click on the diagram.
3. In the dialog box that appears, specify the name and the interaction called.
4. Click **Finish**.

You can specify arguments of an interaction use. An argument is a specific value corresponding to a parameter of the interaction called. In addition, when the argument has been created on the interaction use, you must align it with the interaction parameter called.

To create an argument:

1. Open the properties dialog box of the interaction use.
2. Click the **Characteristics** tab.
3. In the **Arguments** frame, click the **New** button.
   A value specification is created.
   You can rename it and specify its characteristics by opening its properties dialog box.

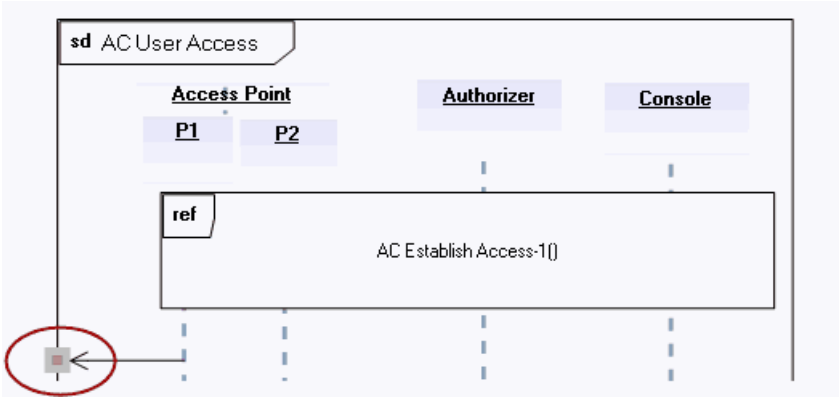
To align the argument with the interaction parameter called:

1. In its properties dialog box of the interaction use, select the **Characteristics** tab.
2. Click the arrow at the right of the **Interaction called** box and select **Modify** .
   A dialog box displays characteristics of the interaction called.
3. For each parameter, click in the value column and select the corresponding value specification.

Gate

A gate is a connection point between a message external to an interaction fragment and a message belonging to this interaction fragment.

Example



To create a gate in the sequence diagram:

1. Click the **Gate** button in the object insert toolbar.
2. Click on the frame outlining the interaction at the point you wish to position the gate.
   The gate then appears in the diagram.

Continuation

A continuation is a syntax means for defining the continuation of sequences of different branches of an Alternatives combined fragment. Continuations are similar to labels representing intermediate points in a control flow.

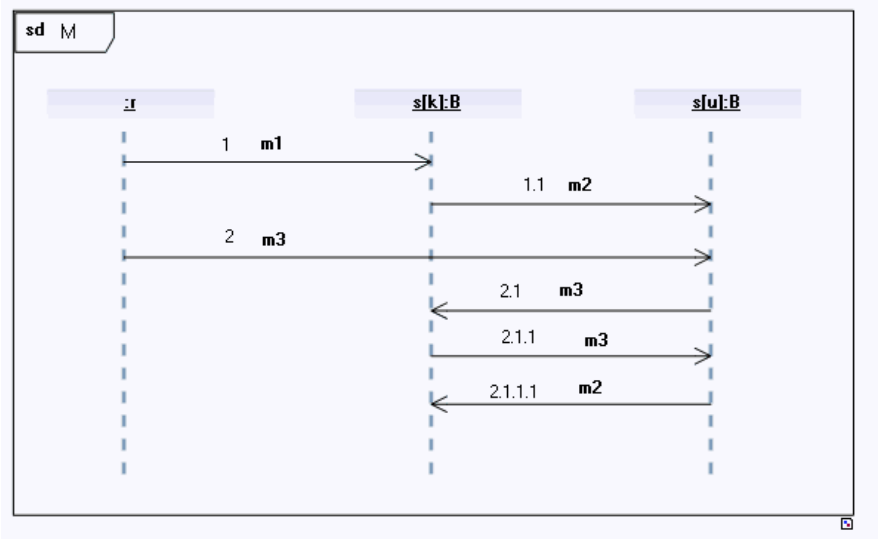
COMMUNICATION DIAGRAM

The communication diagram is a simplified representation of the sequence diagram, concentrating on message exchanges between objects within an interaction.

The sequence and communication diagrams are isomorphic. When a communication diagram relates to an interaction already described in a sequence diagram, it is automatically initialized from the information contained in the sequence diagram.

Example

Sequence Diagram



Communication Diagram

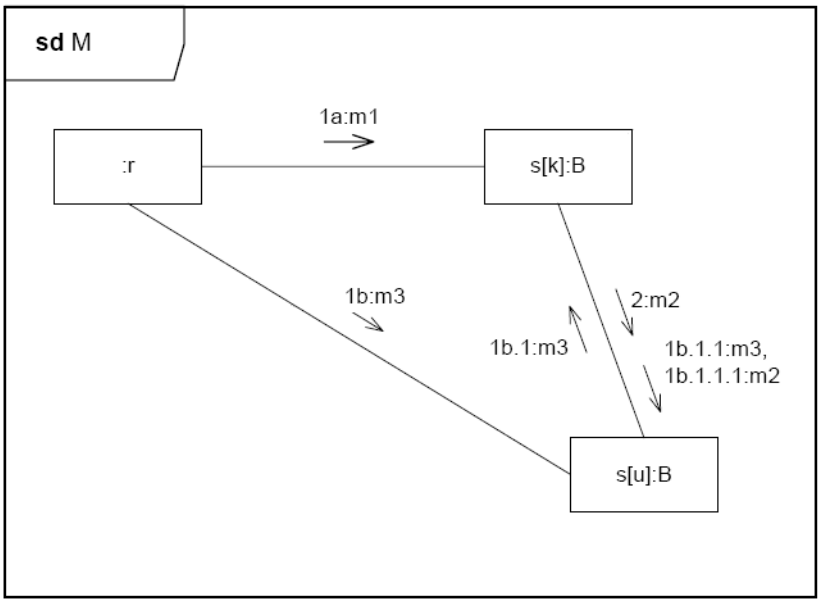
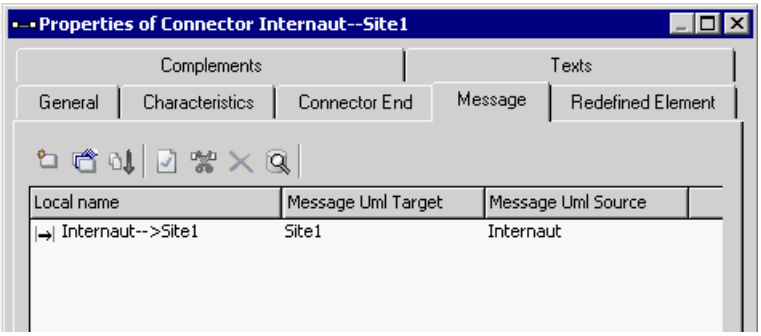


Diagram objects

Communication diagram objects are lifelines and messages transmitted by connectors.

When you connect two lifelines with a connector , the connector creation dialog box proposes messages that may be transmitted.

When the connector has been created, you can associate new messages in its properties dialog box, in the **Message** tab.

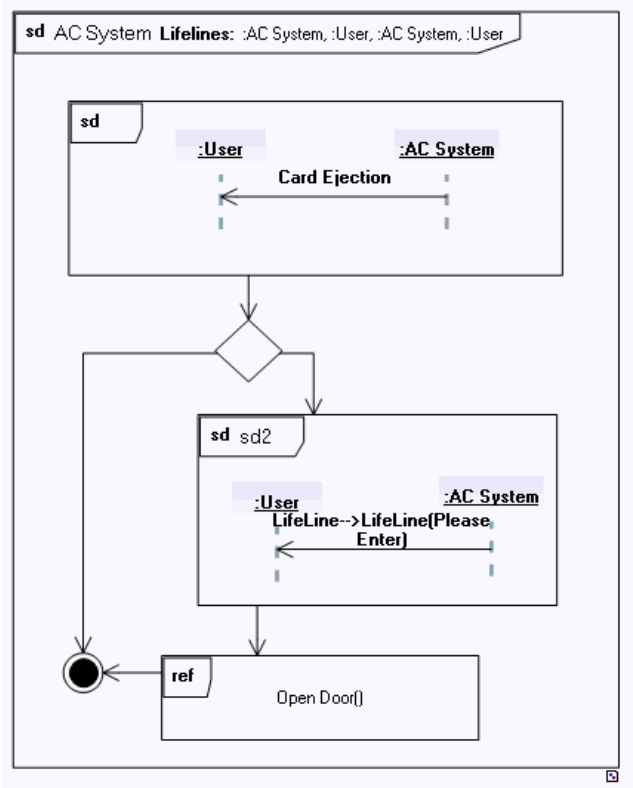


The sequence of messages is given by a sequence number associated with each message. See "Calculating sequence numbers", page 125.

For more details on connectors, see "Connectors", page 95.

INTERACTION OVERVIEW DIAGRAM

The interaction overview diagram describes sequences possible between scenarios previously identified in the form of sequence diagrams. It gives an overview of control flows.



Objects represented in the interaction overview diagram are interactions and interaction uses, lifelines, messages, control nodes and control flows.

THE DEPLOYMENT DIAGRAM

The deployment diagram, which complements the component diagram with hardware resources on which components run

✓ "Presentation of the Class Diagram", page 138.

PRESENTATION OF THE CLASS DIAGRAM

The deployment diagram complements the component diagram. It describes hardware resources (computer, router, etc.) in the system, and indicates distribution of components on these hardware resources.

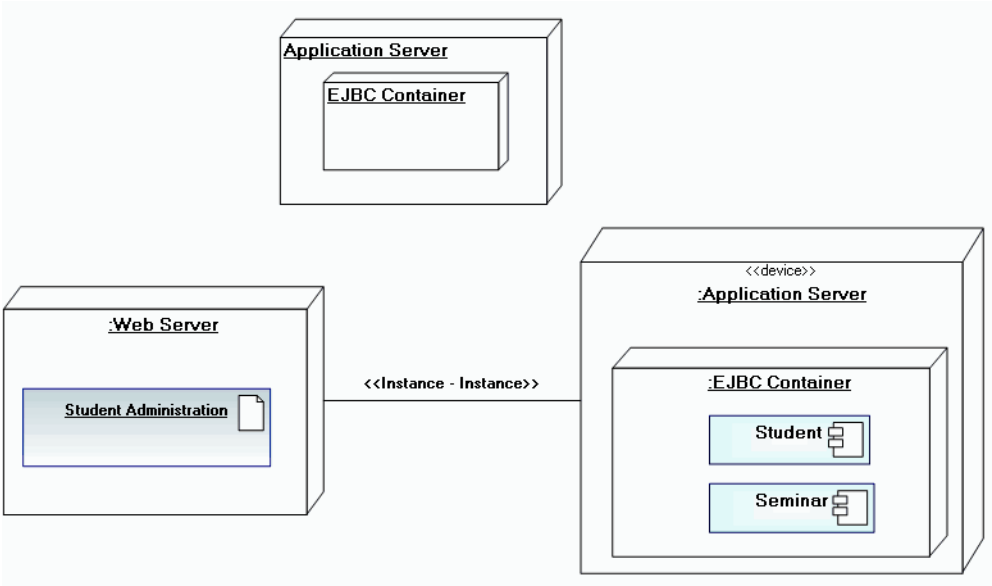
It also describes connections between components or nodes.

This diagram also allows specification of interfaces required and implemented for sequencing of components.

It can be illustrated and supplemented by the addition of node, component or class instances.

You can create a component diagram from a package.

Example of a deployment diagram



Deployment diagram objects

Node

A node is a physical object representing an IT resource, generally with a memory and often with calculation capabilities, on which components can be deployed.

Nodes can comprise other nodes or artifacts. To indicate that a component is assigned to a node, either place the component in the node, or connect the component to the node by a dependency link.

See "Dependency links", page 99.

You can create a node in the deployment diagram using the **Node (UML)** button in the insert toolbar.

Communication path

Connections between nodes are represented by communication paths via which signals and messages are exchanged.

Component

A component represents a modular part of a system that encapsulates its content, and which can be replaced in its environment. A component defines its behavior by means of interfaces that it provides and requires.

One component can be replaced by another if their interfaces conform.

A component can be a software package, program, code unit, etc.

Artifact

An artifact represents a physical information element used or produced by the software development process, or by the deployment or implementation of a system. Example: source files, scripts, executable binary files, development deliverables, word processing documents, electronic messages, etc.

Manifestation

A manifestation is the real physical restoration in an artifact of one or several modeling elements such as components or classes.

The source of a manifestation dependency is an artifact, the target a component or class.

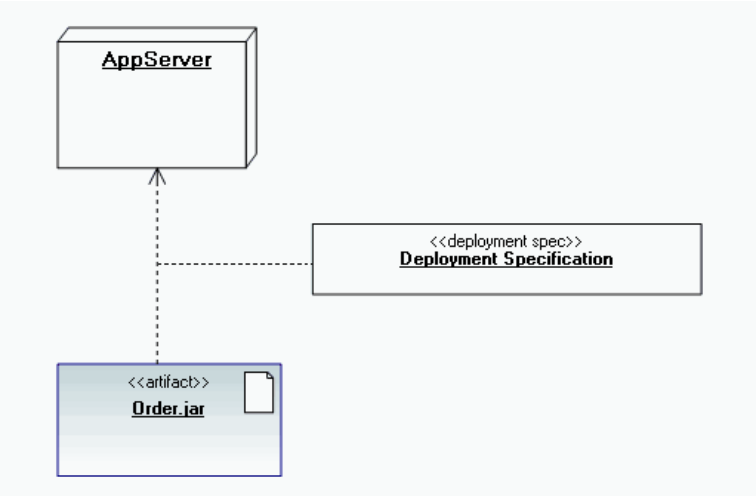
Deployment specification

Deployment specification enables indication of the characteristics that determine execution parameters of an artifact or component deployed on a node.

Configuration

The configuration button enables creation of the link between a deployment specification and a deployment.

Example


AppServer
<<deployment spec>>
Deployment Specification
<<artifact>>
Order.jar

APPENDIX: ATTRIBUTE TYPE

The following points are covered here:

- ✓ "Primitive Types", page 142
- ✓ "Packages and Primitive Types", page 144
- ✓ "Defining New Primitive Types", page 147

PRIMITIVE TYPES

A primitive type is used to group characteristics shared by several attributes. Primitive types are implemented as classes.

Defining a Primitive Type

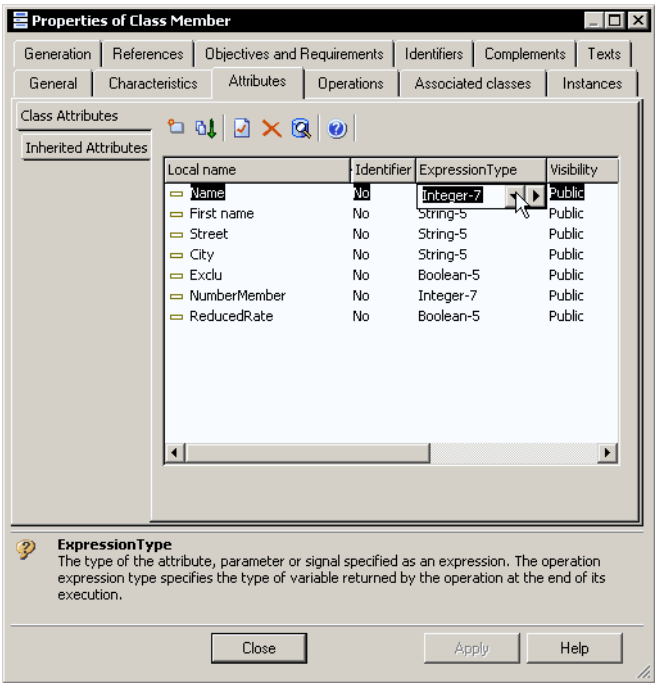
Primitive types are defined in a class diagram.

These are classes for which the following is specified:

- They are of the "Primitive Type" stereotype.
- They are "Abstract" classes because they will not be instantiated.
- They are "Non-persistent" classes. They should not have a corresponding table in the database.

To specify types of class attributes:

1. In the Properties dialog box of the class, select the **Attributes** tab.
2. Click the **Type expression** field and select the attribute type using the arrow.



The following classes are in the standard list:

Alphanumeric types		**Other Information**
M-Char	Alphanumeric string of fixed length	Length
M-Varchar	Alphanumeric string of variable length	
Numeric types		
M-Numeric	Number	Length, decimal places
M-Amount	Amount expressed as currency	Length, decimal places
Date types		
M-Date	Date	
M-Time	Time	
M-Datetime	Date and time	
Binary types		
M-Timestamp	Identification automatically generated from the date and time, expressed in thousandths of seconds since January 1, 1970	
M-Bool	Boolean, equals 0 or 1	
M-Multimedia	Binary string	

PACKAGES AND PRIMITIVE TYPES

Packages

A package partitions the domain and the associated work. It is used to organize elements into groups, particularly use cases and classes. Packages may be nested within other packages. Packages are linked by contractual relationships that define their interface.

The assignment of classes to packages imposes a rigid structure. As a class can belong to only one package, it is necessary to define client/supplier relationships so packages can use classes they do not own when they need to.

This is especially important for primitive type classes, because they will be used to define the attributes of other classes.

Rule: a class can belong to only one package.

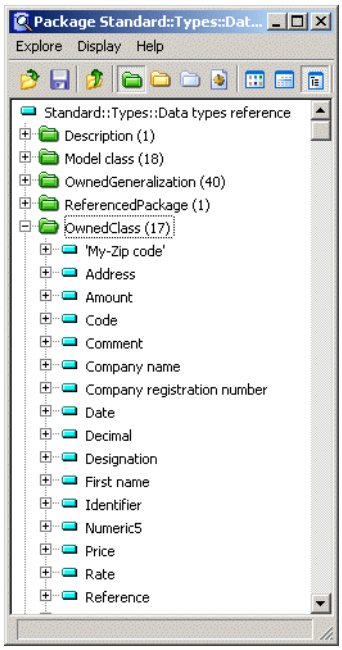
What primitive types are available for typing the class attributes depends on which package the class is in.

The type you can give to class attributes can only be primitive types defined for the package containing the class.

The accessible primitive types are public classes with the "Primitive Type" stereotype, that are contained in or are used by the package or the packages of which it is the client.

You can define a reference package (or several reference packages) containing the primitive types used by the enterprise. All the other packages are declared as clients of the reference package of primitive types.

In the example below, the “Data types reference” package contains the classes “Address”, “Code”, “Date”, etc.



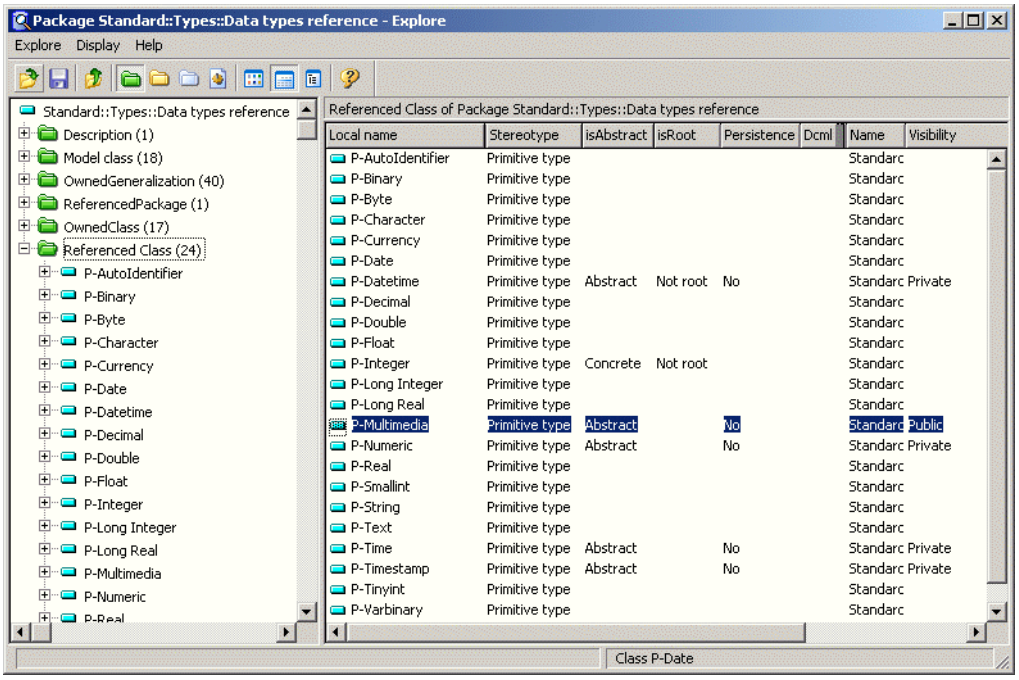
It is referenced by the packages "Library", "Order management", etc.

The class attributes for these packages can be typed using the types “Address”, “Code”, “Date”, etc.

It is also possible to specify directly that a package uses a class contained in another package.

In the example below, the classes “P-Datetime”, “P-Multimedia”, "P-Numeric", etc. are used by the “Data Type Reference” package without being owned by that package.

Of these classes, only “M-Multimedia” is exported by the package for public use.


Package Standard::Types::Data types reference - Explore
Explore Display Help
Standard::Types::Data types reference
Description (1)
Model class (18)
OwnedGeneralization (40)
ReferencedPackage (1)
OwnedClass (17)
Referenced Class (24)
P-AutoIdentifier
P-Binary
P-Byte
P-Character
P-Currency
P-Date
P-Datetime
P-Decimal
P-Double
P-Float
P-Integer
P-Long Integer
P-Long Real
P-Multimedia
P-Numeric
P-Real
Referenced Class of Package Standard::Types::Data types reference
Local name | Stereotype | isAbstract | isRoot | Persistence | Dcml | Name | Visibility
P-AutoIdentifier | Primitive type | | | | | Standarc |
P-Binary | Primitive type | | | | | Standarc |
P-Byte | Primitive type | | | | | Standarc |
P-Character | Primitive type | | | | | Standarc |
P-Currency | Primitive type | | | | | Standarc |
P-Date | Primitive type | | | | | Standarc |
P-Datetime | Primitive type | Abstract | Not root | No | | Standarc | Private
P-Decimal | Primitive type | | | | | Standarc |
P-Double | Primitive type | | | | | Standarc |
P-Float | Primitive type | | | | | Standarc |
P-Integer | Primitive type | Concrete | Not root | | | Standarc |
P-Long Integer | Primitive type | | | | | Standarc |
P-Long Real | Primitive type | | | | | Standarc |
P-Multimedia | Primitive type | Abstract | | No | | Standarc | Public
P-Numeric | Primitive type | Abstract | | No | | Standarc | Private
P-Real | Primitive type | | | | | Standarc |
P-Smallint | Primitive type | | | | | Standarc |
P-String | Primitive type | | | | | Standarc |
P-Text | Primitive type | | | | | Standarc |
P-Time | Primitive type | Abstract | | No | | Standarc | Private
P-Timestamp | Primitive type | Abstract | | No | | Standarc | Private
P-Tinyint | Primitive type | | | | | Standarc |
P-Varbinary | Primitive type | | | | | Standarc |
Class P-Date

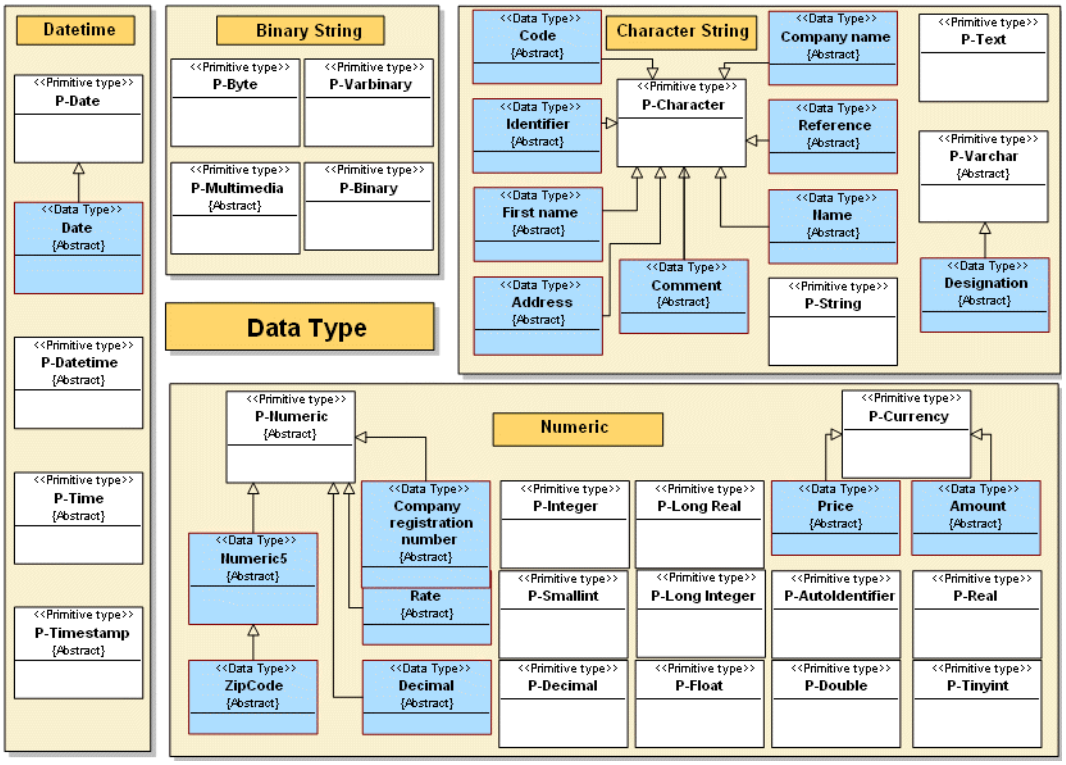
DEFINING NEW PRIMITIVE TYPES

New primitive types can be defined using a class diagram.

Depending on whether classes have been organized into packages, the class diagram can describe:

- A reference database.
- The package of reference types.

You can define your own primitive types by declaring them as subclasses of the standard primitive types, as shown in the example below:

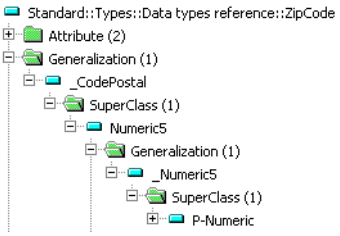


The primitive types defined as subclasses will automatically inherit the characteristics of their superclass. In particular, the datatype conversion rule for the superclass is applied to the subclass.

It is possible to specify a length and a number of decimal places for the subclass. These will be taken into account when generating the data types if they were not already defined for the superclass.

Inheritance can occur at several levels.

In the following example, the primitive type “ZipCode” is a specialization of the “Numeric5” type of length 5, which is itself a specialization of the standard type “P-Numeric”.

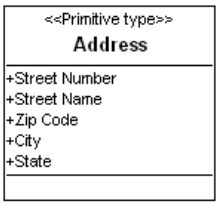


If the new primitive type is not defined directly or indirectly as a subclass of a standard primitive type, the conversion table that maps primitive types to column data types must be updated.

> ☛ *A connection can also be directly defined between a type and the corresponding SQL datatype generated for each target DBMS without using the inheritance mechanism (see "Correspondances entre types pivots et datatypes" dans le guide **HOPEX Database Builder**).*

Compound Primitive Type

You can define a compound primitive type by assigning to it a list of attributes.

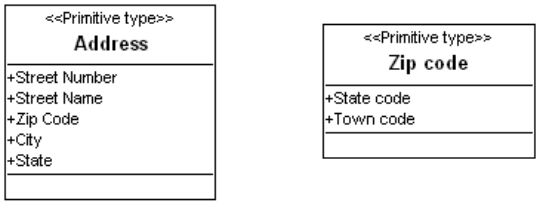


Here the Address type is composed of number, street, zip code, city, and country.

The derivation of the Address attribute will produce these five columns.

It is possible to have several levels of compound types by assigning a compound type to an attribute of a compound type.

For example, the zip code can be broken down into the five main digits and the four-digit extension:



HOPEX XML Schemas

User Guide

HOPEX V2

Information in this document is subject to change and does not represent a commitment on the part of MEGA International.

No part of this document is to be reproduced, transmitted, stored in a retrieval system, or translated into any language in any form by any means, without the prior written permission of MEGA International.

© MEGA International, Paris, 1996 - 2016

All rights reserved.

HOPEX is a registered trademark of MEGA International.

Windows is a registered trademark of Microsoft Corporation.

The other trademarks mentioned in this document belong to their respective owners.

INTRODUCTION

An XML schema is a model that describes the structure of the information contained in an XML document.

> *XML stands for eXtensible Markup Language. This is a language for describing and exchanging structured documents. The first version of XML was adopted by the W3C in January 1998. Users can add new tags to XML. This is why the language is said to be extensible. The ability to add extensions to the language is the key feature of XML.*

A document structure can be defined, and then documents or messages checked against the defined format. Companies that share the same XML document formats can exchange documents and messages over the Internet.

HOPEX UMLenables users to reverse engineer, specify, and generate XML schemas.

HOPEX provides two methods for building XML schemas. The user can work:

- in the XML editor, where the standard XML concepts are found.
- in the diagram editor, where the user can model XML schemas using concepts from the UML standard. The advantage of using UML is that it offers concepts such as inheritance or associations, which are useful when modeling XML schemas.

Whatever the method, it is the user who specifies the XML schemas that can be generated in format XSD (eXtensible Schema Definition).

This document explains how to build XML schemas using **HOPEX UML**, but does not cover the XML methodology in general.

CONTEXTS FOR USING XML SCHEMAS

XML schemas are used in the following contexts:

- ***Publishing informations*** involves numerous XML documents with complex links.
- ***Processing e-commerce transactions***: schema libraries define transactions within market places and between multiple parties.
- ***Control messages and data***: when managing and using network peripherals, data and control messages must be exchanged. Schemas can be used by a server to verify the validity of outgoing messages or by the client to determine what part of the message it understands.
- ***Creating traditional documents***: schemas can play an important role in applications that guide authors as they build their documents. The application can verify that the data entered by the author are correct.
- ***Creating queries***: schemas can assist with formulating and optimizing queries. A database can be created automatically by a schema to inform other systems of what it contains, along with useful queries.
- ***Exchanging metadata***: l'intérêt pour l'échange de métadonnées est grandissant et les schémas XML permettent de faciliter leur interopérabilité.

  Metadata include all information about the source and history of production data, and the processing that the data have undergone in order to be usable in a data warehouse.

CONVENTIONS USED IN THIS GUIDE

☛ *Remarque sur les points qui précèdent.*

📖 *Définition des termes employés.*

☺ *Astuce qui peut faciliter la vie de l'utilisateur.*

🦕 *Compatibilité avec les versions précédentes.*

💣 **Ce qu'il faut éviter de faire.**

Remarque très importante à prendre en compte pour ne pas commettre d'erreurs durant une manipulation.

Les commandes sont présentées ainsi : **Fichier > Ouvrir**.

Les noms de produits et de modules techniques sont présentés ainsi : **HOPEX**.

PRESENTATION OF THIS GUIDE

The **HOPEX XML Schemas** guide comprises the following chapters:

- "Presentation of the Schema Editor", page 7: describes the editor dialog box.
- "Creating an XML Schema", page 19: explains how to create an XML schema in **HOPEX UML**.
- "More XSD Features", page 49: presents concepts particular to XSD.
- "More UML Features", page 59: explains management in the XML schema of UML concepts such as inheritance, association classes and stereotypes.
- "Controls", page 75: describes checks linked to building schemas for each generation format.
- "Components", page 81: describes the stucture of a component and explains its generation.
- "Complete Examples of XML Schemas", page 85: provides XML schema examples.

  The following chapters cover specific cases of generation:
- "XSD Generation", page 105.
- "XSD Reverse Generation", page 129

PRESENTATION

The XML concepts used in **HOPEX UML** are introduced here with Schema Editor. Before building an XML schema in **HOPEX**, it is necessary to pass through several stages.

The following points are covered here:

- ✓ "Prerequisite Conditions", page 8
- ✓ "Definitions", page 9
- ✓ "Example of Schema Created in HOPEX", page 12
- ✓ "Description of the Schema Editor", page 14

PREREQUISITE CONDITIONS

In order to allow the XML schema to reference XSD types, it is necessary to import the "Information Architecture - XsdType" Solution Pack.

Importing the XSD Solution Pack

The "Information Architecture - XsdType" Solution Pack is available in the Utilities\Solution Pack folder of your HOPEX installation directory.

You must decompress it before importing it with **HOPEX Administration**.

For more details, see "Importing Reference Frameworks in HOPEX", page 229.

DEFINITIONS

✓ "Defining a Tag", page 9
✓ "Element", page 10
✓ "Attribute", page 11
✓ "Namespace", page 11

Defining a Tag

Tag definition is the concept that identifies the names of its attributes, the multiplicity of its attributes, and sub-tags contained in the tag. This concept defines a family of tags (we will use UML classes to define tags). Instances of these tags are found in XML documents. Tag definition should be distinguished from tag.

A tag definition is characterized by:

- the list of tag *attributes*.
- The sub-tags potentially nested within the tag, becoming part of its definition.
- Information describing the order of these sub-tags, how often they appear, etc.

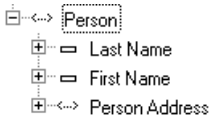
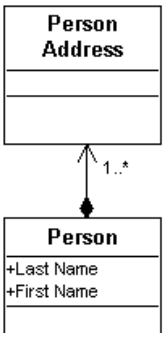
Tag definition example

```
<xsd:ComplexType name="Person">
   <xsd:sequence>
      <element name="Address_Person" type="Address_Person"/>
   </xsd:sequence>
   <xsd :attribute name="Name" Type="xsd:string"/>
   <xsd :attribute name="Name" Type="xsd:string"/>
</xsd:complexType>
```

Tag instance example

```
<Person name="Smith" first name="Peter>
<Address_Person">
   12 rue Cabanis
</Address_Person>
</Person>
```

Here are the two possible representations of this tag definition in **HOPEX**:

Diagram	Diagrams
	

UML modeling

A UML class represents a family of XML tags, not a specific tag in a document

☺ *Through associations, you can reuse a tag definition to define several tags that have the same characteristics in a document. For example, the <WorkAddress> and <HomeAddress> tags can use the same <Address> definition.*

☛ *Throughout this document, look for an explanation of the corresponding UML diagram modeling method in the boxes labeled "UML Modeling".*

Element

An element represents a tag in the schema editor The XML elements of a tag A are defined by the set of tags potentially included in A.

UML modeling

When creating an element, a hierarchical relation appears between the tag definitions, which is why associations between classes are a natural representation.

When you create an *element*, **HOPEX** automatically creates the association between the class declaring the element and the element type.

This means the element is composed of the class + association.

Example

If the "Person" tag definition includes tags based on the "Address_Person" tag definition, the classes "Person" and "Address_Person" defining these tags will be linked by an association.

Properties of the element

To display the properties of the element, right-click the element and select:

- **Type properties** to display the properties of the class defining the element.
- **Role properties** to display the properties of the association role corresponding to the element.

☛ *When the editor is in "standard level" mode, the pop-up menu presents only the properties of the role. See* *"Schema Editor Display Parameters", page 14*.

Attribute

The XML attributes of a tag are pairs of strings in the format name-value. The second string is contained within quotes. XML attributes serve to characterize the tag. Example: <person name= 'Smith'/>.

Namespace

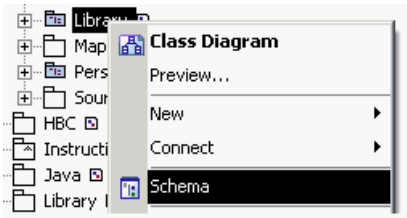
The namespace is a set of predefined terms (set of tag definitions, elements and attributes of a document). Inside this space, two tags are not allowed to have the same name.

☛ *See* *"Accessing Namespace Types", page 15*.

EXAMPLE OF SCHEMA CREATED IN HOPEX

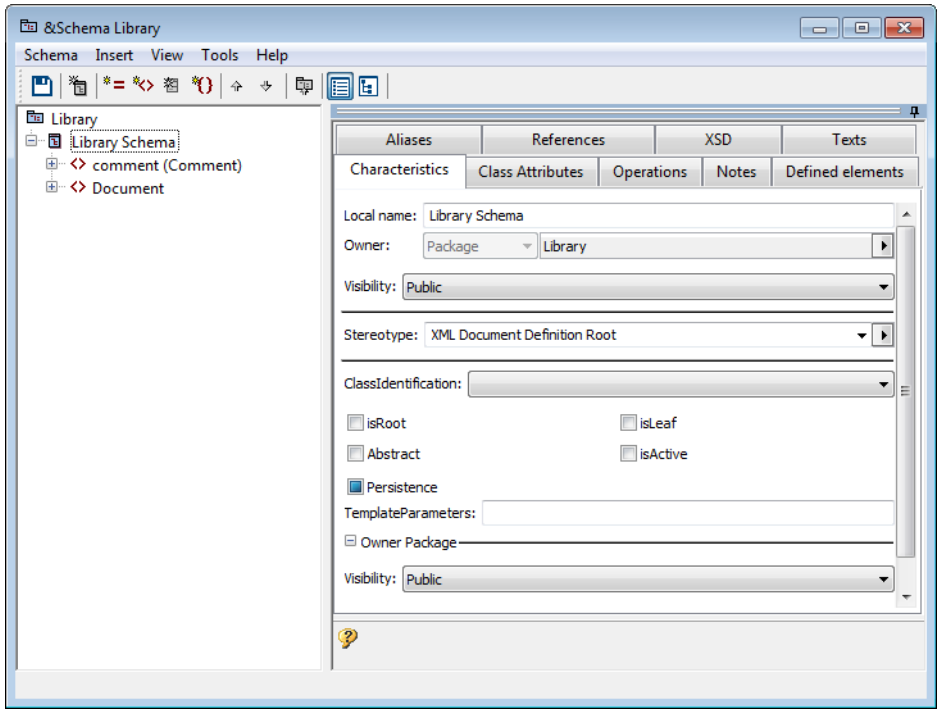
To display examples of schemas created in **HOPEX**:

1. In the **Main Objects** navigation window, open the "Data Exchange" package.
2. In the pop-up menu of the "Library" package, select **Schema**.



The Schema Editor is opened.

3. Unfold the schema by clicking on the +.
4. Click the **Details** button in the toolbar to display the properties dialog box.



This example shows a document structure containing lists of authors, books, and collections.

The document structure is represented by the schema named "Library".

To show the content of the lists, the authors, books, and collections are included as "elements" (or "sub-tags" <…>).

Therefore the schema is defined as a tree where one element includes other elements, which in turn may include other elements.

Attributes (represented by **=**) are defined for elements.

This document structure can also be represented in a diagram:

UML modeling

To open the diagram representing the "Library" schema:

- Select **Class Diagram** in the pop-up menu of the "Library" package.
  The class diagram opens.

The hierarchical representation is modeled by the composite associations.

DESCRIPTION OF THE SCHEMA EDITOR

The left part of the schema editor is a navigator, displaying the composition of the schemas.

The right part of the editor shows the properties of the element selected in the navigator.

✓ "Schema Editor Display Parameters", page 14
✓ "Accessing the Complete Properties Dialog Box for Schema Elements", page 15
✓ "Accessing Namespace Types", page 15
✓ "Generating a Diagram from the XML Schema", page 16

Schema Editor Display Parameters

By default, the schema editor is in standard level mode.

To access the advanced functionalities required for the construction of XML schemas:

- In the **Tools** menu, select **XML Expertise Level**.
- In the **View** menu, click **Toolbar** and verify that **Filterbar** is displayed.

Buttons enable you to modify display of elements in your schema.

Display attributes: displays attributes of elements.

Display of inherited classes: allows display of the tag definitions that a tag definition has inherited.

Display types: displays the namespace types.

Display of elements: displays attributes of elements.

Keep editor in front

Tree only: hides the right-hand part of the editor.

Details: displays properties of elements.

Refresh: refreshes the window to take into account new elements.

Accessing the Complete Properties Dialog Box for Schema Elements

The editor properties dialog box can be accessed by clicking the **Details** button in the navigator. This properties dialog box is partial, displaying only the properties of the selected element.

To, access the entire Properties dialog box:

- Click the **View** menu .
- Clear **Filter Properties Pages**.

You can also right-click the element in the XML navigator and select:

- **Type properties:** to display the properties of the class defining the element.
- **Role properties**: to display the properties of the association role corresponding to the element.

☛ *These two commands appear when you are in XML Expertise mode.*

Accessing Namespace Types

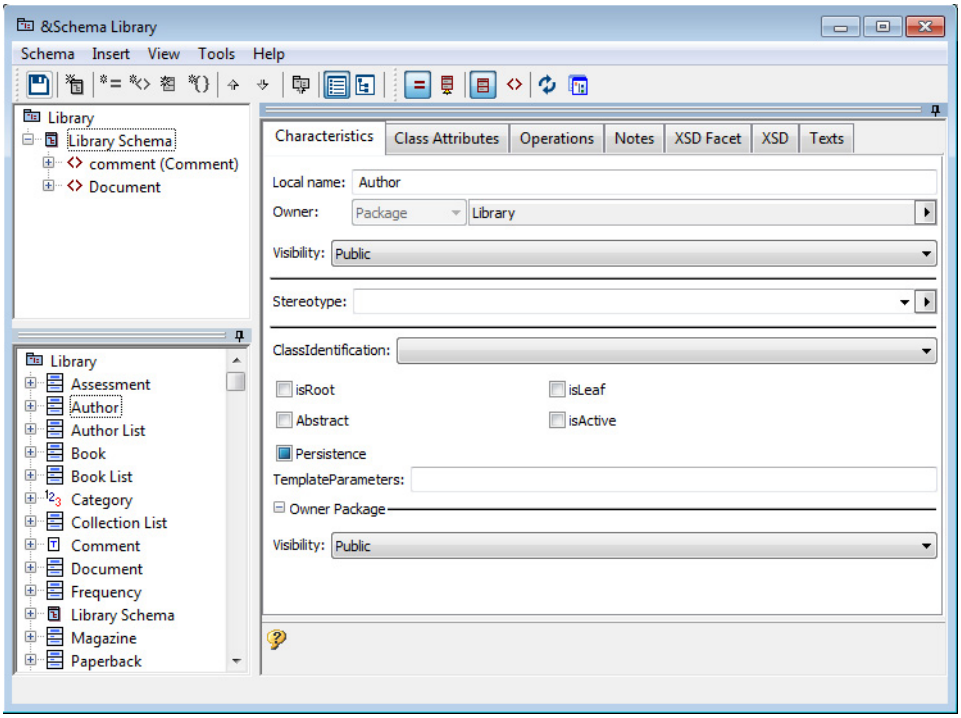
The schema editor allows you to display or not the types of *namespace*.

The namespace is a set of predefined terms (set of tag definitions, elements and attributes of a document). Inside this space, two tags are not allowed to have the same name.

To access all elements in the namespace:

- Click the **Display of types defining elements** button.

The types are displayed in a list under the XML schema.



When you click a type, its properties dialog box appears in the right part of the editor.

☛ *See also "Defining a Namespace", page 20.*

Generating a Diagram from the XML Schema

You can generate a UML diagram from the XML Schema, if you want to represent the schema in a diagram.

To generate a diagram from the XML schema:

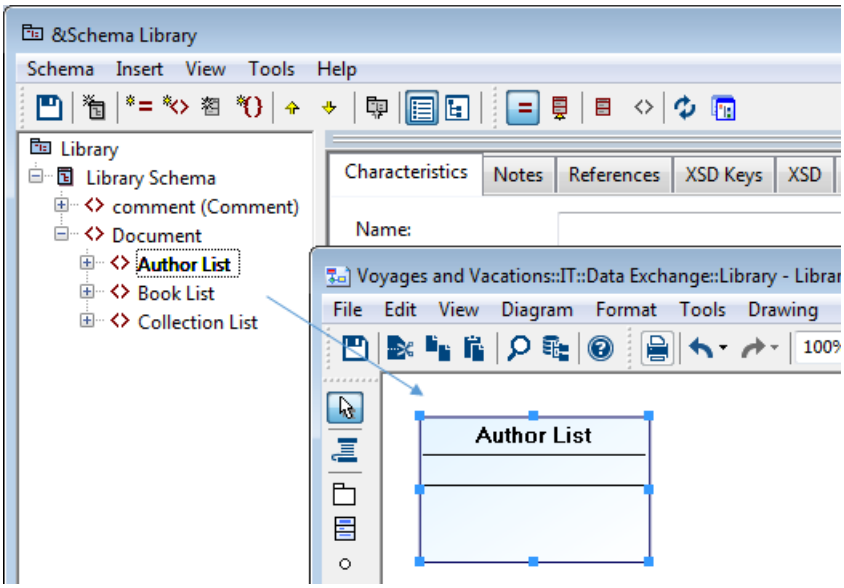
❱ Right-click the schema name and select **New Generate a diagram**.

You can generate two types of diagram:

- **a 1 level class diagram**: describes the first level of classes associated with any association classes, the first level of inheritance and the first level of implemented interfaces
- **a fanout diagram**: describes ***n*** levels of classes associated with any association classes (recursion stops if a class C does not belong to the same package as the described class but if c is included in the diagram), the first level of inheritance and the first level of implemented interfaces.

You can also build the diagram yourself.

- Right-click the schema and select **New Diagram**.
- In the dialog box that opens, select "Class Diagram" and click on **Create**. The diagram that opens is empty.
- You can drag the desired elements from the XML Navigator directly into the class diagram.



The buttons on the left are used to create the classes, associations, etc. representing the XML schemas. See the **HOPEX UML** guide for the functionalities of UML modeling.

From this diagram you can access the properties of schema elements.

- Right-click the desired object in the diagram.
- Select **Properties**.

Its dialog box appears. You can specify and modify the properties of the element.

CREATING AN XML SCHEMA

There are three essential steps in defining the structure of a document:

- Create and name the schema (the document structure).
- Define the content of the document, which means including the schema elements
- Define the attributes of the schema elements

The points covered here are:

✓ "Defining a Namespace", page 20
✓ "Opening the Schema Editor", page 22
✓ "Creating a Schema Class", page 23
✓ "Defining an Alias for a Package", page 25
✓ "Import and Inclusion of Other Schemas", page 27
✓ "Connecting a Tag Definition to a Schema Class", page 29
✓ "Defining Elements in an XML Schema", page 30
✓ "Defining attributes", page 33
✓ "Groups", page 37
✓ "References", page 42
✓ "Using Packages for Classification", page 48

Defining a Namespace

To make reference to predefined types such as XSD in your XML schema, you have to first import their respective libraries. See "Importing the XSD Solution Pack", page 8.

Before building an XML schema, you should first define the namespace, which will serve as a "workspace" for the user. When the namespace has been created, you can then open the XML schema editor.

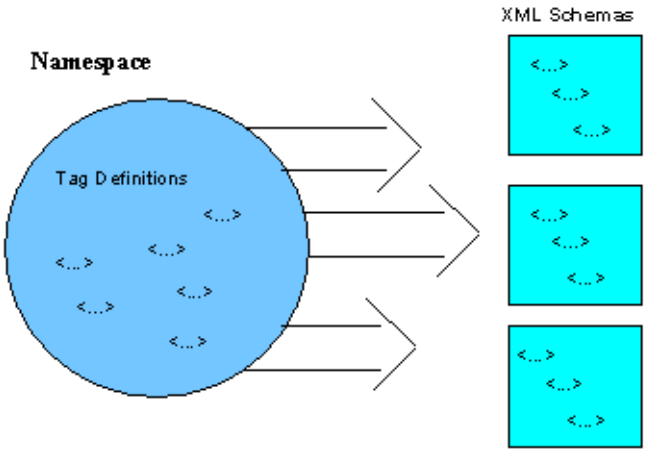
> *The namespace is a set of predefined terms (set of tag definitions, elements and attributes of a document). Inside this space, two tags are not allowed to have the same name.*

Example:

```
In the "Library" namespace there can be only one "Book" tag,
but it can be reused as many times as required.
```

The advantage of defining a *namespace* is that types created in the namespace can be reused in another namespace.

> *The types are then prefixed by their namespace name, to avoid collisions when they are imported into another namespace.*

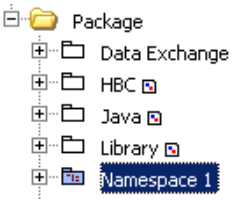


In **HOPEX UML**, namespaces are represented by packages.

To create a namespace:

1. Click the **Main Objects** window of the main navigator in the **HOPEX** bar.
2. Right-click **Package**.
3. Select **New**> **Package**.
4. In the dialog box that appears, enter the name of the future namespace and click **OK**.
   A new package is created in the list of packages.
5. Select **Properties** in the pop-up menu of the new package.
   The properties dialog box for the package appears.

6. In the **Characteristics** tab, in the **Stereotype** field, select "XML Document Definition".
7. Click **OK**.
   The namespace is created in the list of packages:



You can now open the schema editor.

Opening the Schema Editor

To open the schema editor:

- In the pop-up menu of the namespace you just created, select **Schema**.

The XML schema editor opens. By default, it displays the "Standard level" and filters a certain number of characteristics.

To access the advanced functionalities required for the construction of XML schemas:

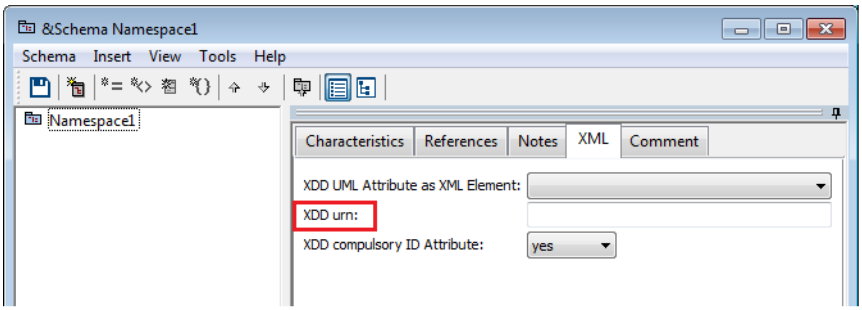
- In the schema editor, click the **Tools** menu and select **XML Expertise Level**.

Specifying the URN for a Namespace

The URN is the technical identifier for a namespace. It uniquely describes the namespace.

To specify the *URN* for a namespace:

1. In the schema editor navigator, click the namespace.
2. In the right part of the editor, select the tab for the generation language you are using and enter the URN for the package in the **[XDD] urn** box.

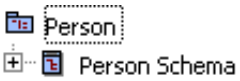


CREATING A SCHEMA CLASS

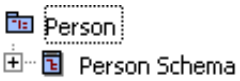
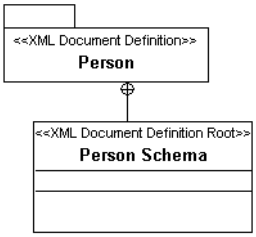
Before creating an XML schema, you must first define a namespace. See "Defining a Namespace", page 20.

To create an XML schema:

1. Click the icon for **create schema class** button in the XML schema editor toolbar.
2. In the dialog box that appears, enter the name of the schema and click **OK**.
   The new schema appears in the navigator:



Result

Diagram	Diagram
	

UML modeling

To create a new XML schema:

1. Click the **Class** button and name the class in the dialog box that appears.
   The new class appears in the diagram.
2. Open the properties dialog box for the class (right-click the class and select **Properties**).
3. In the **Characteristics** tab, click on the arrow in the **Stereotype** list box and select "XML Document Definition Root".

This class represents the schema.

☺ *Each XML schema is represented in **HOPEX** by a class of the XML Document Definition Root stereotype. When you create a new XML schema in the XML schema editor (not in the diagram), **HOPEX** automatically creates a class of the XML Document Definition Root stereotype to represent the schema.*

☺ *If the class was not created in an XML Document Definition package, the package that owns the class is automatically modified to be a package of stereotype XML Document Definition.*

You can also select **New > Schema** from the pop-up menu for a package. In this case, the stereotype is automatically assigned to the class and the package (if the package does not already have an ancestor of the XML Document Definition stereotype).

if you create the class in the diagram and set the stereotype manually, the owner package is not modified to have the "XML Document Definition" stereotype. We therefore recommend using one of the two procedures described above.

DEFINING AN ALIAS FOR A PACKAGE

An alias is a short name given by the user to a URN (the identifier for a namespace).

Aliases are used to prefix the names of elements of a namespace when the elements are used in a namespace other than their own. The aliases identify which namespace the elements belong to when they are used in a different namespace context.

If you connect a type belonging to another namespace to your schema, the name of the type will contain the alias that you defined.

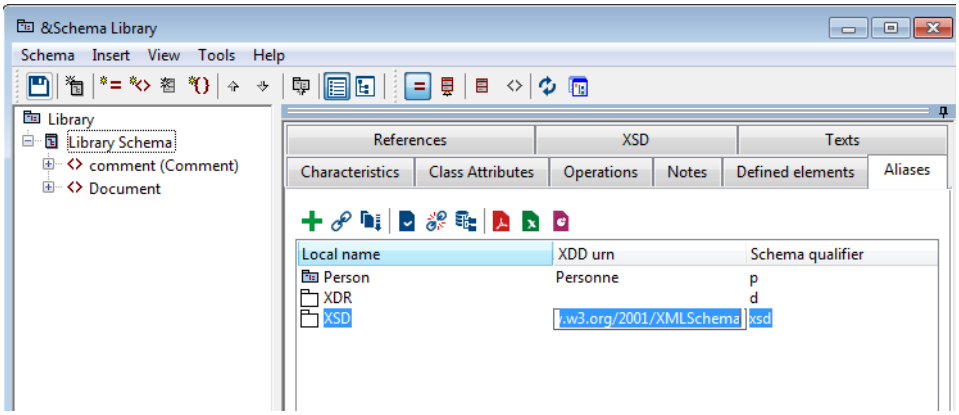
```
Example : d:string

The schema references the "string" type represented by
alias "d".
```

This alias is defined in the context of the schema, and is not part of the actual namespace. A different user can define their own alias for the package as desired.

To define the alias for a package:

1. Click the schema where you plan to connect a type from another package.
2. In the properties dialog box, select the **Aliases** tab.
3. Click the package to which the type belongs.
4. In the **urn** field, enter the urn.
5. In the **Schema Qualifier** field, define the alias.



To connect a tag from another namespace:

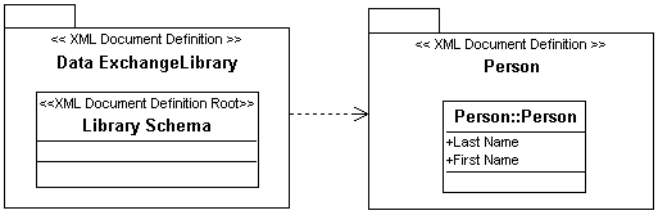
1. Click the **Connect** button.
2. In the selector dialog box, select **Namespaces with urn**.
3. In the dialog box that opens, select the namespace containing the types, elements and attributes to be included in the schema.
4. Click **Add**.

Example

The "Library" schema wants to use the "Person" tag definition. To do this, it connects to the "Person" namespace, which contains the "Person" tag definition.

The alias given to namespace "Person" (if "Person" is the URN for the namespace) to which "Person" belongs is p.

The "Person" type will be prefixed by "p" in the Library schema.



In the document, the result is:

```
<Library xmlns:p="Person">
<p:Person/>
</Library>
```

IMPORT AND INCLUSION OF OTHER SCHEMAS

As in any programming language, it is important to be able to program in a modular manner. In this way the programmed elements are classified by concept. Several concepts can be assembled in the same library.

Standard libraries are also provided in XML. Each of these libraries contains hundreds of schemas, each treating a particular concept.

Import and inclusion of these schemas is treated differently in XSD.

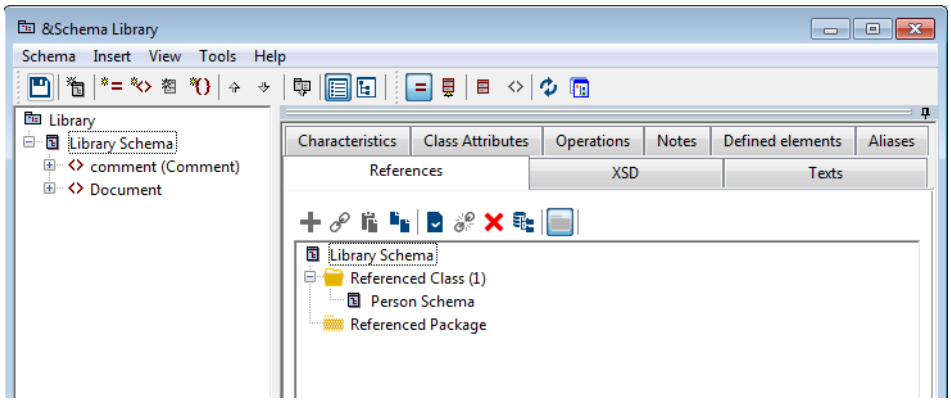
XSD

You can however reference types belonging to other namespaces. To do this, you must reference the namespaces using the xmnls attribute. This attribute defines an alias on a namespace.

However, this reference is not sufficient to use types. They must be included or imported into the current schema. Distinction is made between:

- Inclusion: this consists of including schemas that belong to the same namespace as the current schema.
- Import: this consists of importing schemas that that do not belong to the namespace of the current schema, or complete namespaces.

In the following example, the "Library" schema imports the "Person" schema that belongs to another namespace.



To import the schema of another namespace:

1. Select the schema into which you want to import another schema.
2. In its properties dialog box, select the **References** tab.
3. Select "Referenced Class" or "Referenced Package" depending on whether you want to import a particular schema or all schemas contained in another namespace.
4. To do click the **Connect** button.
5. Find the schema or package and click **OK**.

UML modeling

To reference a package from a class diagram:

1. Select **View** > **Views and Details** and select the **Schema** check box.
2. In the objects tool bar, click the **Link** button
3. In the diagram, click the schema class and connect it to the package to be referenced.
4. A dialog box will ask you to select the type of link you wish to create. Select **Referenced Package** and click **OK**.

☛ *In the same way you can connect a class to a schema class. This is a link of "Referenced class" type.*

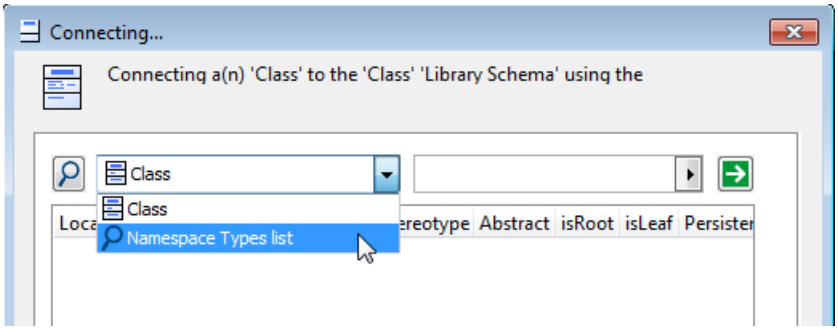
Connecting a Tag Definition to a Schema Class

When the same namespace contains several schemas, it is necessary to connect the tag definitions to the schema class to which they belong.

> ☛ *This operation is optional if the namespace only consists of a single schema. Also, elements created in the schema editor are automatically connected to the schema.*

To connect a tag definition:

1. Click the schema to which you want to connect the tags. The properties tabs for the class appear in the right part of the editor.
2. In the **Defined Elements** tab, click the **Connect** button.
   The Query dialog box appears:
3. Select **Namespace types list**.



4. In the dialog box that opens, select the types to be connected to the schema and click **Connect**.

UML modeling

- Open the properties dialog box of the **Schema** to which you wish to connect the elements (pop-up menu/**Properties**).
- Proceed as described above.

DEFINING ELEMENTS IN AN XML SCHEMA

An element represents a tag in the schema editor The XML elements of a tag A are defined by the set of tags potentially included in A.

Creating an Element

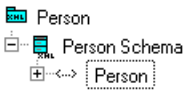
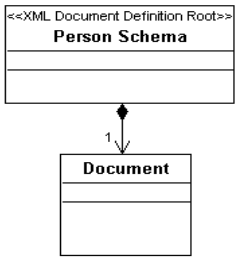
To create a new *element*:

1. Click the schema where the element is to be created, then click the **create element** button in the toolbar.
   The **Add element** dialog box opens.

   ☛ *Content of this dialog box varies according to the mode selected. To obtain all the properties of an element, you must select "XML Expertise Level", accessible from the **Tools** menu of the schema editor.*

2. Enter the name of the element, specify the characteristics and click **Create**.

Result

Diagram	Diagram	XML Document
		<Document>

Characteristics of an Element

When creating an element, several fields should be completed:

Multiplicity

The multiplicity of an element determines the number of times the corresponding tag appears. All the usual values can be used for the multiplicity, including:

0: the element does not appear (useful for eliminating an inherited definition),

1: the element must appear, but only once,

*****: the element is optional and can be repeated multiple times,

1...*: the element must appear at least once, and may appear more than once.

To specify the multiplicity:

- Select the multiplicity using the arrow.

Element Type

In this box, you can define the element type (or element definition).. By default, the name of the type is the same as that of the element.

Basic type (optional)

This is the type inherited by the element type.

Inheritance (optional)

Corresponds to the inheritance type: restriction or extension.

Local

Local determines if the tag definition is created locally, that is at the level of the tag definition containing the element.

If you do not select this box, the tag definition is created globally, that is at the level of the namespace. It can therefore be reused.

Access by reference

If you select this check box, the element becomes a reference. See "References", page 42.

The element is created under the schema in the navigator:

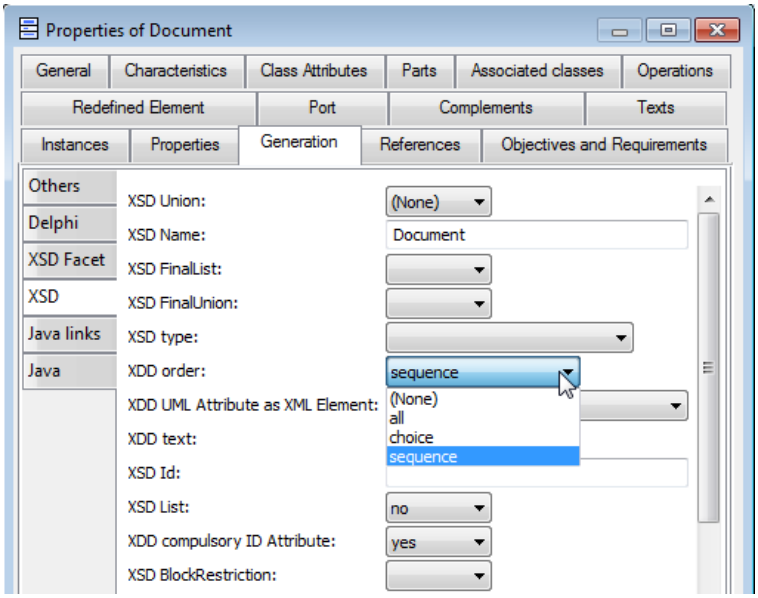
Ordering Contained Elements

To specify the order in which contained elements are generated:

1. Right-click the element to be ordered and select **Type properties** in its pop-up menu.

   ☛ *You must be in "XML Expertise Level" for this command to appear.*

   Its dialog box appears.
2. In the **Generation** tab, select the **XSD** sub-tab and select a value for the **XDD Order** list box using the arrow.
   - "All": type elements appear zero or one times and in any order. "All" groups are only authorized at the highest level of a type and must appear alone. In addition, they can only contain elements.
   - "Sequence": the tags should appear in order, and the number of times specified by the multiplicity of the element.
   - "Choice": only one of the sub-tags can appear. The number of times is specified by the multiplicity.



UML modeling

❱ Open the **Properties** dialog box of the class for which you wish to order the component classes and proceed as described above.

DEFINING ATTRIBUTES

The XML attributes of a tag are pairs of strings in the format name-value. The second string is contained within quotes. XML attributes serve to characterize the tag. Example: <person name= 'Smith'/>

An attribute can only contain information related to its value.

Creating an Attribute in an XML Schema

To create a new *attribute*:

1. Select the element for which you wish to create a new attribute, then click the **Create Attribute** button in the editor toolbar.
   The properties dialog box for the attribute appears.
2. Enter the name of the attribute and click **OK**.
   The attribute is created.

Result

Diagram	Diagram	XML Document
Person Last Name	Person +Name	<person name= "Smith"/>

UML modeling

To add attributes:

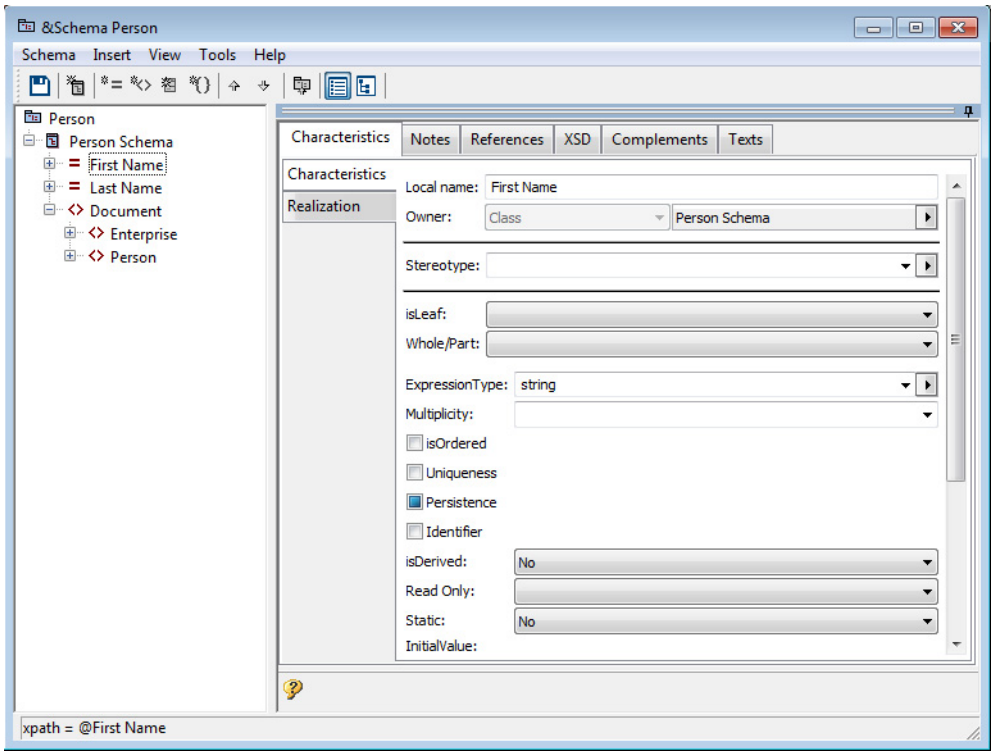
1. Open the **Properties** dialog box of the class for which you wish to create an attibute (pop-up menu/**Properties**).
2. In the **Attributes** tab, click the **New** button
3. Enter the name of the attribute and click OK.

Attribute Characteristics

To define attribute characteristics:

- Select the attribute on the left of the schema editor. The properties for the attribute will appear on the right.

  ☛ *You can also access the properties dialog box by right-clicking the attribute and selecting **Properties**.*



Attribute type

The **ExpressionType** allows you to define the type of attribute. The list that appears corresponds to common package types and referenced namespace types.

Initial value

Each attribute can have a default value. When a tag does not define an optional attribute, if the attribute has a default value, this value will be assigned to it automatically.

To define a default value:

- Open the properties dialog box of the attribute.
- Click the **Characteristics** tab.
- In the **Initial Value** box, enter the default value.

Example

The collection class has a name attribute with an initial value of "SimonSchuster".

XSD

```
<xsd:ComplexType name="Collection">
    <xsd:attribute name="name" default="SimonSchuster"
use="required">
      <xsd:SimpleType>
      <xsd:restriction base='xsd:string'>
        <xsd:enumeration value='SimonSchuster'/>
        <xsd:enumeration value='RandomHouse'/>
        <xsd:enumeration value='Bantam Books'/>
      </xsd:restriction>
    </xsd:simpleType>
   </xsd:attribute>
  </xsd:ComplexType>
```

Multiplicity

The multiplicity of an attribute defines the number of occurrences of the attribute in a class.

In a schema, an attribute can only appear once (a **Person** has only one name for example). Multiplicities that can be specified are:

0: attribute not allowed,

0..1: optional,

1: required (default value).

- In the **Multiplicity** text box, select **0**, **0**...**1 or 1**.

The 0 multiplicity is useful when a class inherits an undesired attribute: using the overloading function, it is possible to set the multiplicity of the attribute to 0 and therefore eliminate the attribute. See *"Other Features Related to Inheritance", page 64*.

Static

It is possible to specify if the attribute can be modified. **Static** specifies if the attribute can take specific values for each instance of the class or take one value characterizing the entire class.

- "Yes": the attribute has a value that characterizes the entire class. The attribute "Telephone number length" for the "USA Client" class is 10 digits.
- "No": the attribute can take a different value for each class instance. For example, the "Telephone number" attribute has a different value for each instance of the "Client" class.

Example

For the **Collection** attribute, if the field is set to:

- "Yes": the value of the **Collection** attribute cannot be modified.
- "No": the "SimonSchuster" value can be changed to "RandomHouse" for example
- "Add only": this parameter is not used in the context of XML schemas. If you select it, the attribute is considered as changeable.

UML modeling

To define the properties of an attribute:

1. Open the properties dialog box for the class (right-click it and select **Properties**) and select the **Attributes** tab.
2. Select the attribute concerned and click the **Properties** button : The properties dialog box for the attribute appears.

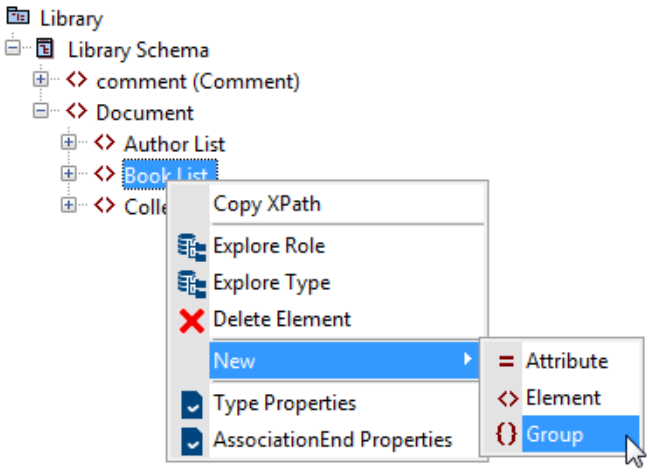
GROUPS

A group is a set of elements or attributes. It is usefule to constitute groups to order the elements they contain.

Creating a Group in an XML Schema

To create a new *group*:

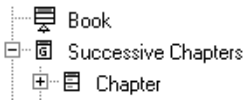
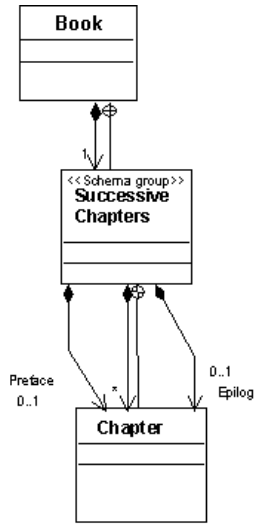
1. Right-click the element for which you want to create a group in the XML navigator and select **New** > **Group**.



2. In the dialog box that opens, enter the name of the group.
3. Click **Create**.

> ☛ *The group creation dialog box varies according to the mode selected. To obtain all the properties of a group, you must select "XML Expertise Level", accessible from the **Tools** menu of the schema editor.*

Example

Diagram	Diagram	XML Document
		<Paperback> <Preface/> <Chapter> </Chapter> ... <Epilog> </Epilog> </Paperback>

UML modeling

A group is represented by a class of the **Schema Group** stereotype.

- Create a Class
- Open its properties page.
- Click the **Characteristics** tab.
- In the **stereotype** field, select "Schema Group" .

Attribute Group Template

It is possible to define templates for attribute groups. This type of group does not contain any elements but contains a certain number of attributes and attribute subgroups. The concept of order does not exist for attribute groups.

These groups appear at schema level and are then reused within tag definitions by referencing. This avoids redundant definitions.

Element Group Template

As for attributes, it is possible to define element group templates. The aim is to avoid definition redundancy since inheritance between groups does not exist. These

groups are defined at schema level. They contain element sub-groups (all, sequence, choice). They are referenced within tag definitions.

Ordering the Elements in a Group

The elements of a group can be ordered.

To order the elements:

- Select the group in the schema editor navigator.
- In the right part, select the **Characteristics** tab.
- In the **Order** list box, select one of the following values using the arrow .
  - "All": type elements appear zero or one times and in any order. "All" groups are only authorized at the highest level of a type and must appear alone. In addition, they can only contain elements.
  - "Sequence": the elements must appear in the order specified by the Order link attribute between the **Schema Group** class and the association roles.
  - "Choice": only one of the elements appears.

You can also order elements in the creation of a group dialog box. See "Creating a Group in an XML Schema", page 37.

or:

- In the schema editor, right-click the group to be ordered and select **Type properties**.
  Its dialog box appears.
- In the **Generation** tab, in the generation language subtab (**XSD**) complete the **XDD Order** list box using the arrow.

The value of the **XDD Order** parameter and the multiplicities of the different sub-elements define in what order the sub-elements appear within the group, and how often.

The following table gives some examples:

XDD Order	Multi-plicity for A	Multi-plicity for B	Multi-plicity for C	Comment	Example
Choice	1	*	1	One A tag or (0 or more B tags) or one C tag	<X> <A/> </X>
Sequence	1	1	*	One A tag followed by one B tag followed by 0 or more C tags.	<X> <A/> <B/> <C/> <C/> </X>
all	1	1	1	One tag of each in any order.	<X> <B/> <C/> <A/> </X>

UML modeling

- Display the properties dialog box for a class of the **Schema Group** stereotype and proceed as described above.

Using a Group

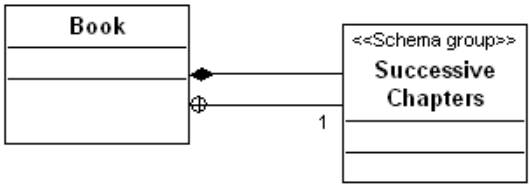
Groups can be used as classes when declaring a set.

In XSD, a special tag is created (<sequence>, <choice>, <all>).

Example

Tags defined for the paperback class declare a group instance (multiplicity of 1). Then, to avoid needlessly overcrowding the schema package, the successive chapters class is declared locally to the paperback class (symbol ().

Local: this characteristic determines whether the group is created locally to the tag definition or group containing the element, or whether it is created globally (at namespace level).



XSD

```
  <xsd:ComplexType name="Book">
     <xsd:sequence>
    <xsd:element name="Summary" type='Summary'
minOccurs="1" maxOccurs="1"/>
    <xsd:sequence minOccurs="1" maxOccurs="1">
      <xsd:element name = 'Preface' type="Preface"
minOccurs="0" maxOccurs="1"/>
      <xsd:element name = 'Chapter' type="Chapter"
minOccurs="0" maxOccurs="*"/>
      <xsd:element name = 'Epilogue' type="Epilogue"
minOccurs="0" maxOccurs="1"/>
    </xsd:sequence>
   </xsd:sequence>
   <xsd:attribute name="ISBN" type="xsd:ID" use="required"/
>
   <xsd :attribute name="Author" type="xsd:IDREF"
use="required"/>
   <xsd:attribute name="Collection" type="xsd:IDREFS" />
   <xsd:attribute name="category" type="category"
use="required"/>
  </xsd:ComplexType>
```

REFERENCES

References are used to create a relationship between two tags without having to include the referenced tag. For example, if a document contains a list of persons, you can refer to the persons declared in the list without having to include the actual declaration. This eliminates redundancies and facilitates document maintenance.

Creating a Reference in an XML Schema

In the schema editor, the reference can only be created from an element.

To create a *reference*:

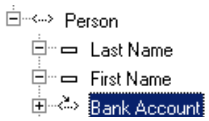
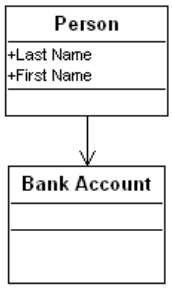
- Select the **Access by Reference** check box in the create element dialog box.

☛ *See "Creating an Element", page 30.*

or:

- In the navigator, click the element that is to be defined as a reference.
- In the right part of the editor, in the **Characteristics** subtab, select the **Access by reference** check box.

Result

Diagram	Diagram	XML Document
		<Person Account="1306" Name="Smith"/> <BankAccount id="1306"/>

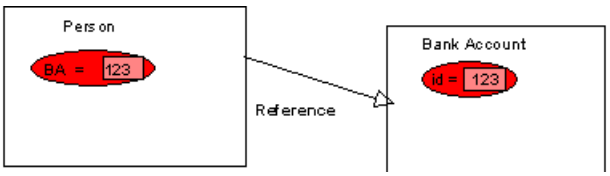
Operating principle

The element defined as the reference will automatically have an attribute to identify it: the id attribute, of type ID.

> ☛ *This type is defined by XML and is therefore available in all the schema languages.*

The tag definition referring to this element will automatically define an attribute of type **IDREF** or **IDREFS** (depending on the multiplicity), whose value will be the value of the **id** attribute for the referenced element.

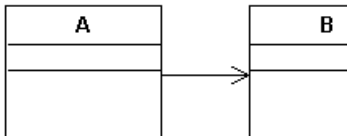
By default, the **id** attribute is added to all tag instances of the tag definition.



> ☛ *Whether this identifying attribute is added is determined by the value of the parameter **XDD compulsory ID attribute**. The default value (**yes**) indicates that the attribute is to be added.*

Example

When a class A references a class B, an attribute of type **IDREF** (or **IDREFS** depending on the multiplicity of the reference) is added to class A. The value of this attribute must correspond to the value of the **ID** attribute for tag B.

Diagram	XML Document
	<A id="25", B="123"/> <B id="123"/>

UML modeling

The reference is represented by an association comparable to that used for declaration of an element. The only difference is in the nature of the association: it is not a composition.

- Create an association between the referenced class and the class that references it.
- Right-click the referencing role and select **Aggregation** > **No** (default value).

Characteristics of a Reference

It is possible to define certain characteristics of the ID attribute for the referenced element:

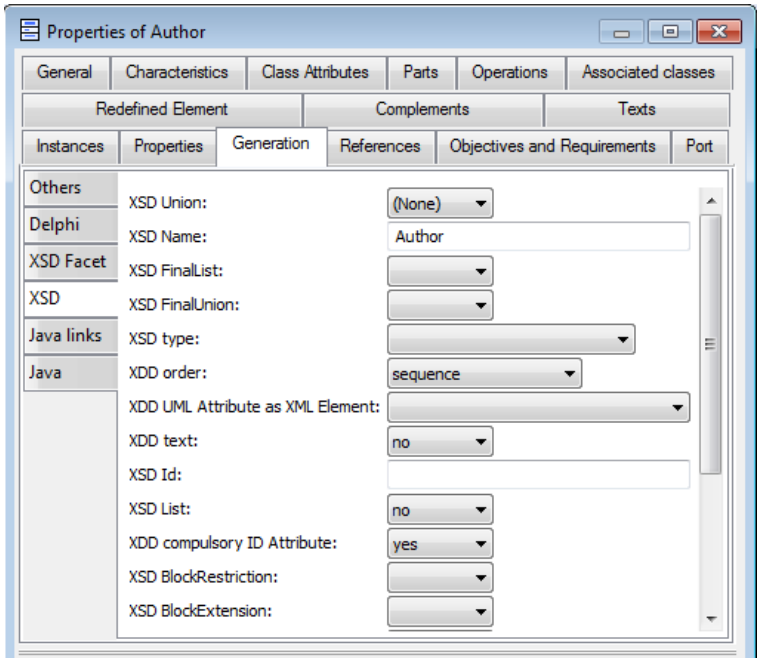
1. Right-click the referenced element and select **Type properties** from its pop-up menu.

   ☛ *The "Type properties" are the properties of the class defining the element.*

   "Role properties" are the properties of the association role corresponding to the element.

   See "Element", page 10.

2. Select the **Generation** tab and then the sub-tab for the generation language you are using.



3. Assign values to the parameters by clicking on their fields and using the arrow to select a value.
   - **XDD Compulsory ID Attribute** parameter: determines whether the attribute is added to the tag definition.

     ☛ *In XSD, an ID type attribute created on the type has priority over an ID attribute defined in generation parameters. Use of this type of reference in XSD is not recommended.*

UML modeling

the value of this attribute may be defined globally for one of the ancestor packages of the class. It is then reflected in all classes owned by the package and its sub-packages. Its value can be redefined locally.

- **XDD ID Attribute Name** parameter: defines the name of the attribute (must be entered manually).
- **XDD ID Attribute Required** parameter: defines whether the value of the attribute must be defined in the document. "No" is selected by default.

Creating a bidirectional reference

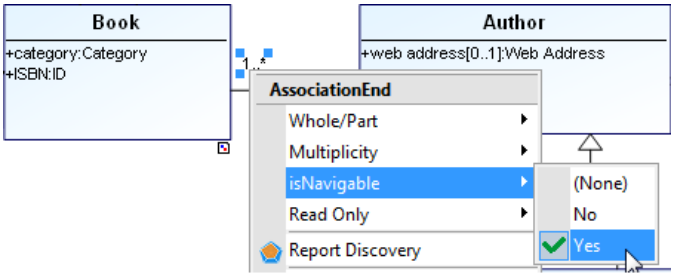
It is possible to reference two tag definitions with the same association.

Simply create a bidirectional association (navigable in both directions) between the two classes representing the schema and the referenced element.

You must also specify multiplicities for the two roles (by default, the multiplicity value is *).

To create a bidirectional reference:

1. Open a diagram by right-clicking the schema and selecting **New > Class diagram**.
2. Drag the two elements from the schema editor into the diagram.
3. Right-click the non-navigable role of the association and select **isNavigable** > **Yes**.



The arrow disappears from the link: the reference has become bidirectional.

4. Save and close the diagram.

Function

In a bidirectional association, an attribute of type **IDREF** (or **IDREFS**) is generated for every tag based on the two connected types. In the instance document, the value of this attribute references the **ID** attribute of the other tag.

Example:

In the example below, each book references its authors, and each author references his/her books.

In the generated source, only the attributes used to reference books from an author are indicated in bold. The attributes used for the reverse reference are easily spotted.

XSD

```
<xsd:complexType name="Book" abstract="false"
mixed="false">
    <xsd:sequence>
      <xsd:element name="Summary" abstract="false"
minOccurs="1" maxOccurs="1" type="Summary">
      </xsd:element>
      <xsd:element ref="comment"/>
    </xsd:sequence>
    <xsd :attribute name="category" use="required">
      <xsd:simpleType>
        <xsd:restriction base="xsd:anyType">
          <xsd:enumeration value="Practical"/>
          <xsd:enumeration value="Science Fiction"/>
          <xsd:enumeration value="Literature"/>
          <xsd:enumeration value="Music"/>
          <xsd:enumeration value="Educational"/>
          <xsd:enumeration value="Children"/>
        </xsd:restriction>
      </xsd:simpleType>
    </xsd:attribute>
    <xsd:attribute name="ISBN" type="xsd:ID"
use="required"/>
    </xsd:attribute>
    <xsd:attribute name="BibliographicReferences"
use="required" type="IDREF"/>
    <xsd:attribute name="Author" use="required"
type="IDREF"/>
  </xsd:complexType>
```

Referencing an Element

It is possible to define an element in the schema. This element can then be referenced within a tag definition. The purpose is to avoid redundant definitions. In this way, we can for example define a "comment" element at schema level and reuse this element in the different schema tag definitions.

To reference an element in the schema editor:

1. Select the element in the navigator.
2. In the right part of the editor, select the **References** tab.

3. Select **Referenced element** and click the **Connect** button.
4. Query the element to be referenced and click **OK**.

UML modeling

To reference an element from a class diagram:

1. Select **View** > **Views and Details** and select the **Schema** check box.
2. In the objects tool bar, click the **Link** button
3. In the diagram, click the referencing role and connect it to the referenced role.
4. A dialog box will ask you to select the type of link you wish to create. Select **Referenced element** and click **OK**.

Referencing an Attribute

It is possible to define an attribute at schema level. This attribute can then be referenced within a tag definition. The purpose is to avoid redundant definitions.

To reference an attribute in the schema editor:

1. Select the attribute in the navigator.
2. In the right part of the editor, select the **References** tab.
3. Select **Referenced attribute** and click the **Connect** button.
4. In the dialog box that opens, find the attribute to be referenced and click **OK**.

Referencing a Group

When a diagram group is defined, its aim is to be referenced.

☛ *See "Attribute Group Template", page 38.*

To reference a group in the schema editor:

1. Select the group element in the left part of the schema editor.
2. In the right part, select the **References** tab.
3. Select **Referenced element** and click the **Connect** button.
4. In the dialog box that opens, find the group to be referenced and click **OK**.

USING PACKAGES FOR CLASSIFICATION

Packages that are not of **XML Document Definition** stereotype can be used:

- either to classify namespaces
- or to partition a namespace

Use of packages that are not of **XML Document Definition** stereotype has no effect on the semantics of the generated schemas.

In this way you can create new non-XML Document Definition packages in your schema, and insert into them elements or namespaces without the packages being generated.

Take care, since the creation of new packages can be a source of errors at the time of reverse engineering.

MORE XSD FEATURES

The XML or XSD (XML Schema Definition) schema is a standard defined by the W3C. Based on XML, XSC enables definition of XML document structure.

The aim of a schema is to define an XML class of document, and therefore the term "document instance" is often used to define a document that is valid in relation to certain criteria.

In order to allow the XML schema to reference **XSD** types, it is necessary to import the corresponding library. See "Importing the XSD Solution Pack", page 8.

- ✓ "XSD: Tag Definitions", page 50
- ✓ "Constraints", page 55

Tag Definitions

Simple Definitions

Simple tag definitions correspond to tags typed without attributes or elements.

In XSD, the user can define simple types that restrict standard types. There are therefore three types of definition for XSD simple types

Union definition

A simple type can be a union of types. This can be useful when several countries use different formats to define a particular concept. For example, postal code is defined differently in France and the USA.

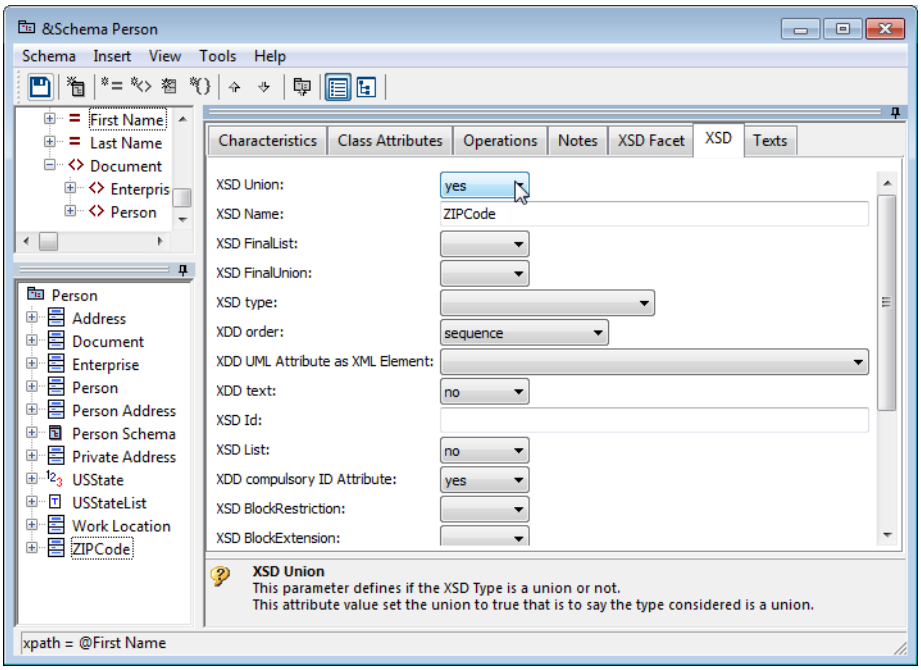
```
<xsd:simpleType name="USState">
  <xsd:restriction base="xsd:string">
    <xsd:enumeration value="AK"/>
    <xsd:enumeration value="AL"/>
    <xsd:enumeration value="AR"/>
    <!-- and so on ... -->
  </xsd:restriction>
</xsd:simpleType>

<xsd:simpleType name="zipUnion">
  <xsd:union memberTypes="USState">
     <xsd:simpleType>
       <xsd:list itemType="xsd :int"/>
     </xsd:simpleType>
   </xsd:union>
</xsd:simpleType>
```

To define a union of types:

1. In the schema editor, select **View** > **Type**.
2. In the navigator, select the class (type) for which you want to create a union of types (in this case "postal code").
3. In the **Properties** dialog box, select the **Characteristics** tab.
4. In the **Stereotype** field, select "Structure" .
5. Select the **XSD** tab.

6. In the **XSD Union** field, select "yes".

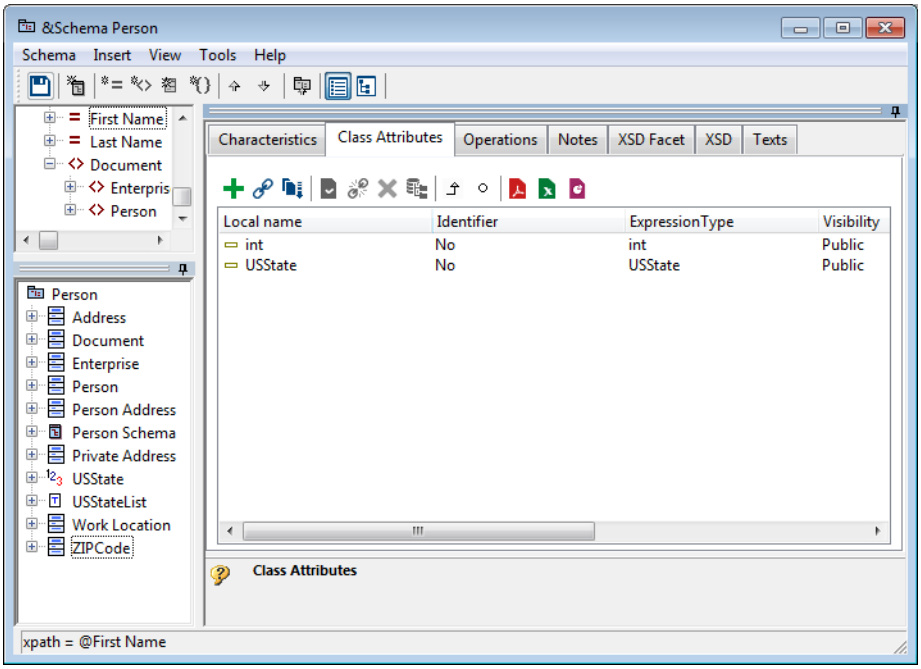


Types defined in the union are shown as class attributes.

To define the types that particpate in the union:

1. In the navigator, select the class representing the union.
2. In its properties dialog box, select the **Attributes** tab.
3. Click the **New** button.
4. Create your attributes. It is preferable to give them the same name as the union types.

5. In the **Expression Type** column, select the types defining the union. In this case, these are the "int" and "USState" types.

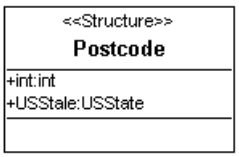


UML modeling

To create a union of types in a diagram:

1. Right-click the class and select **Properties**.
2. In the **Characteristics** tab, select the "Structure" **stereotype**.
3. Select the **Generation** tab and then the **XSD** subtab.
4. In the **XSD Union** field, select "yes".
5. Then define the types of union in the same way.

In our example, the "PostalCode" class is a union of the types "int" and "USState" These types appear in the form of attributes.



Restriction definition (facets)

A simple type can be a restriction of another simple type or a standard XSD type. By restriction, we mean "Enumeration", the limitation of the length of a character chain, definition of an interval for values, etc.

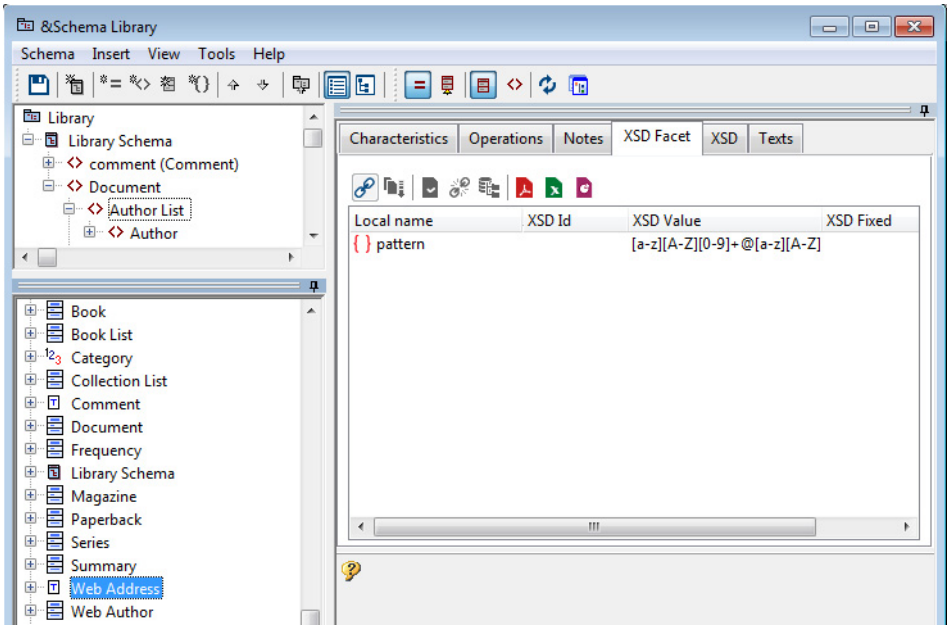
```
<xsd:simpleType name="SKU">
  <xsd:restriction base="xsd:string">
    <xsd:pattern value="\d{3}-[A-Z]{2}"/>
  </xsd:restriction>
</xsd:simpleType>
```

In this example, we define a sku type. This is a string that must begin with 3 figures followed by a dash then 2 ascii characters.

The facets are modeled by constraints on class. Each facet corresponds to an occurence of a constraint.

To define a facet on a type:

1. In the schema editor, select the class (type).
2. In its properties dialog box, select the **XSD facets** tab.
3. Click the **Connect** button
4. Select the list of facets and click **OK**.
5. In the list that appears, select the facet that you want to connect to the class and click **OK**.
6. In the "XSD Value" column, enter the value of the facet.



Particular case of enumeration

To model an enumeration:

1. Open the properties dialog box of the class.

2. In the **Characteristics** tab, select the "Enumeration" **stereotype**.
3. Click **OK**.
4. Reopen the properties dialog box of the class.
   A new tab, **Literal Value** is now present.
5. Enter in this tab the values of the enumeration.

See also "Enumeration Stereotype", page 72.

"List" type definition

A simple type can also be defined as a simple type list. This allows definition of lists or tables that can take as a value a list of elements of simple type.

```
<xsd:simpleType name="USStateList">
    <xsd:list base="USState">
    </xsd:list>
  </xsd:simpleType>
```

To define a type as a list of values:

1. In the navigator of the editor, select the type in question.
2. In its properties dialog box, click the **Characteristics** tab.
3. In the **Stereotype** field, select "Expression" .
4. Select the XSD tab.
5. In the **XSD List** field, select "yes".
   Refresh the properties dialog box of the type and return to the **Characteristics** tab.
6. In the **Expression-Type** field, select the type of value that you want to define for the type. For example, if you want the type to be a list of integers, select "int".

Complex Definitions

Complex tag definitions correspond to tags typed with attributes and elements.

In XSD, we distinguish types that contain attributes only (types with simple content) and those that contain both attributes and elements types with complex content).

We can apply inheritance and polymporphism rules to these types. See "Inheritance", page 60.

CONSTRAINTS

We can define constraints on elements or tags. These constraints are applied to the attributes or to the content of these tags. Constraints defined on tags are uniqueness and setting to correspondence by means of primary and foreign keys.

Uniqueness

Uniqueness enables characterization of an element in a unique way. Uniqueness is modeled by a constraint on the role of a class. Its stereotype is "XSD Uniqueness".

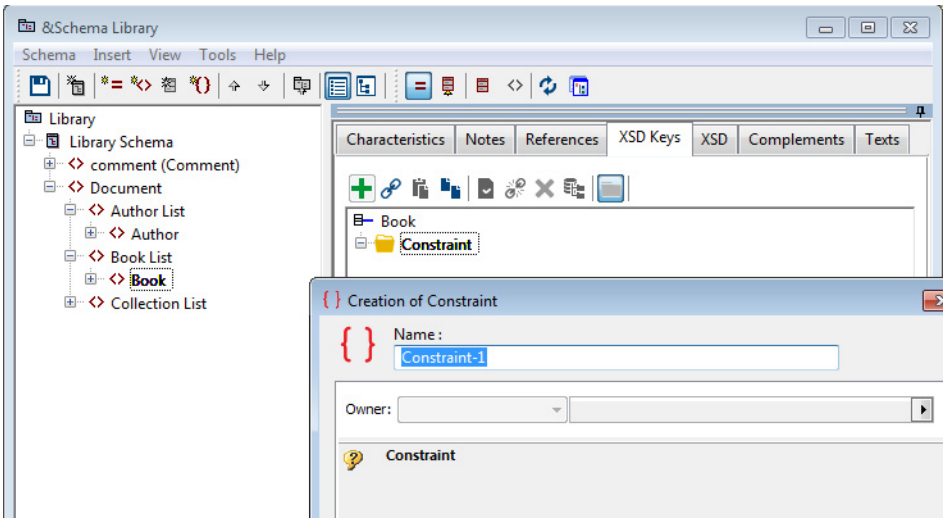
To define uniqueness of an element:

From the schema editor:

- Select the element in the navigator.
- In its properties dialog box, select the **XSD Keys** tab.

To create a constraint:

- Select "Constraint".
- Click the **New** button.
- In the dialog box that opens, enter the name of the constraint and click **OK**.



To connect an existing constraint:

- Select "Constraint".
- To do click the **Connect** button.
- In the dialog box that appears, find and select the name of the constraint and click **Connect**.

The constraint appears in the properties dialog box.

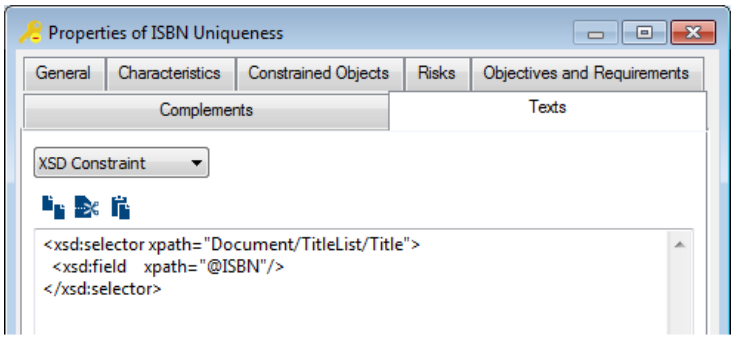
To assign the "uniqueness" type to the constraint:

- Right-click the constraint.
- Click **Properties**.
- In the **Characteristics** tab, select the "**XSD Uniqueness**" stereotype.

Then specify to what the uniqueness refers:

- In the properties dialog box of the constraint, select the **Texts** tab.
- Select "XSD Constraint".
- Specify the elements or attributes that define the uniqueness of the element (field) as well as the path of these elements or attributes (selector). See the "Field" and "selector" tags defined by W3C schemas. XSD schemas use the xpath to characterise an attribute.

In our example, the "Title" class is uniquely defined by the "ISBN" attribute.



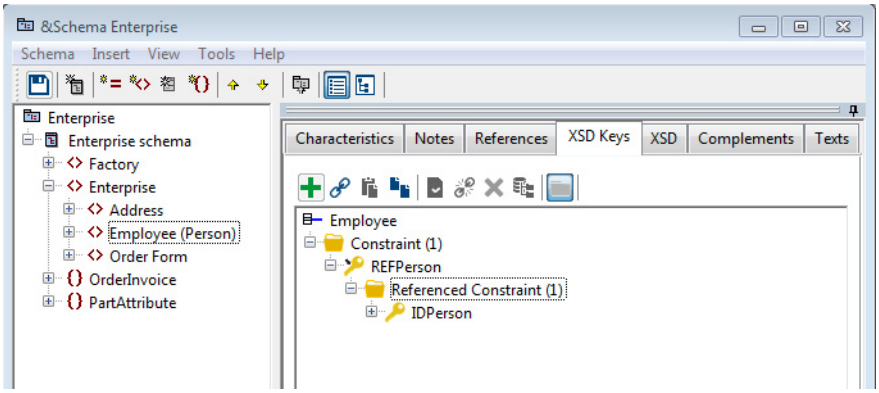
Primary Key and Foreign Key

You can define a uniqueness constraint (primary key) and map this constraint with one or several foreign keys. In this way, elements containing this foreign key cannot exist unless their key takes its value from among those proposed by the primary key.

The primary key is modeled by a constraint of "XSD Key Reference" stereotype. The foreign key is modeled by a constraint of "XSD Key Reference" stereotype. To characterize the fact that a foreign key depends on a primary key, this foreign key is linked to the primary key.

Example:

In the "Enterprise" schema, the "Employee" element defines a "REFPerson" secondary key. This key references the "IDPerson" primary key.



Uniqueness of the secondary key defines a uniqueness constraint on an element. Attributes of this element must have same values as the attributes of elements carrying the primary key.

To connect a secondary key to an element:

1. Select the element in the schema editor navigator.
2. In its properties dialog box, select the **XSD Keys** tab.
3. Select "Constraint".
4. To connect an existing constraint, click the **Connect** button and find the constraint.
   To create a constraint, click the **Create** button and assign the "XSD Key Reference" stereotype to the constraint.
5. Then open the properties dialog box of the constraint.
6. Select the **Text** tab.
7. Select "XSD Constraint" and add a uniqueness text.
8. Click **OK**.

Then indicate to which primary key the secondary key is connected.

- Under the name of the secondary key, select Referenced Constraint".
- Click the **Button** and find the primery key to be referenced.

MORE UML FEATURES

Here are discussed the additional features available when you use UML concepts to model a schema:

The points covered here are:

- ✓ "Inheritance", page 60
- ✓ "Other Features Related to Inheritance", page 64
- ✓ "Association Classes", page 70
- ✓ "Class Stereotypes", page 72

INHERITANCE

Inheritance is a link between two classes, by which the child class inherits all the attributes and elements of the parent class.

Creating an Inheritance Link between Two Tag Definitions

You can create an inheritance link between two tag definitions:

From the class diagram:

- Drag the two types you want to connect by an *inheritance* link into a class diagram. See "UML Modeling".

Or in the creation dialog box of an element:

- In the **Element type** field, specify the type of the new element.
- In the **Repository type** field, select the repository type (the ancestor) of the element type.
- In the **Inheritance** field, select "Extension" or "Restriction".

Inheritance by extension: the element contains the elements and attributes of the basic type but can also contain others.

Inheritance by restriction: the element has exactly the same number of elements and attributes as the basic type but certain of these are restricted (by multiplicity for example).

UML modeling

- Click the **Generalization** button and connect the daughter class to the mother class.

In a schema modeled in UML, any concept represented by a class can inherit a concept of the same kind:

- a simple type can inherit a simple type
- a complex type can inherit a complex or simple type

A complex type is a type containing either elements (composite class) or attributes, or both.

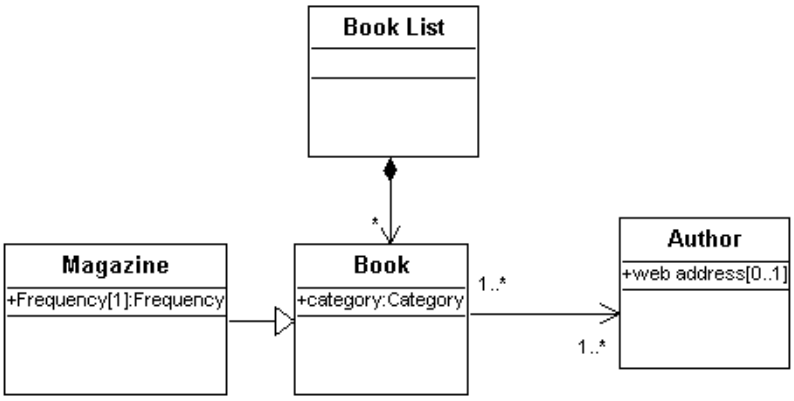
A simple type is a type that contains neither sub-elements nor attributes. This being so, a complex type is necessarily an element type and an attribute type is necessarily a simple type

A class B that inherits a class A can be used to define tags containing the attributes of A and of B.

In the same way, elements defined by the A associations will be added to the elements defined in B.

Example

The "Magazine" class inherits the "Book" class.



The source codes (XSD generated below show that "Magazine" elements have the attributes **Frequency** and **Category**.

There is also an "Author" attribute accessible in a "Magazine" tag. This attribute is inherited from the "Book" class.

XSD

```
<xsd:SimpleType name="FrequencyMagazine">
   <xsd:restriction base='xsd:string'>
  <xsd:enumeration value="monthly"/>
  <xsd:enumeration value="weekly"/>
  <xsd:enumeration value="semimonthly"/>
  <xsd:enumeration value="quarterly"/>
</xsd:restriction>
</xsd:simpleType>
<xsd:ComplexType name="Magazine">
<xsd:complexContent>
     <xsd:extension base='book'>
      <xsd:attribute name="Frequency"
type="FrequencyMagazine" use="Required"/>
   </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
```

Using Inherited Classes

The concept of inheritance is used to define instances corresponding to instances of the inherited class, but with more details.

This being so, if class B inherits A, instances of B can be considered special instances of A.

In consequence, in an XML document, tags A can be substituted by tags B. To view inheritance in the schema editor:

- Click the **Display Inherited Classes** button and unfold the parent class to view the classes it inherits

The inherited classes are represented by the icon .

> *At the time of inheritance, all sub-elements in the namespace are generated. To prevent their generation, you need simply declare the element as abstract. See* *"Creating Abstract Classes", page 67.*

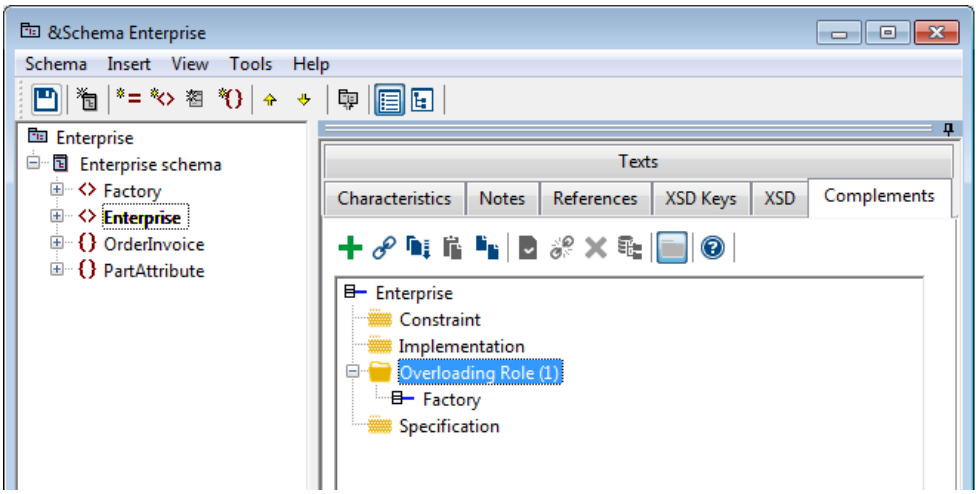
Substitution (XSD)

In XSD, it is possible to specify directly which elements are to be substituted.

Example:

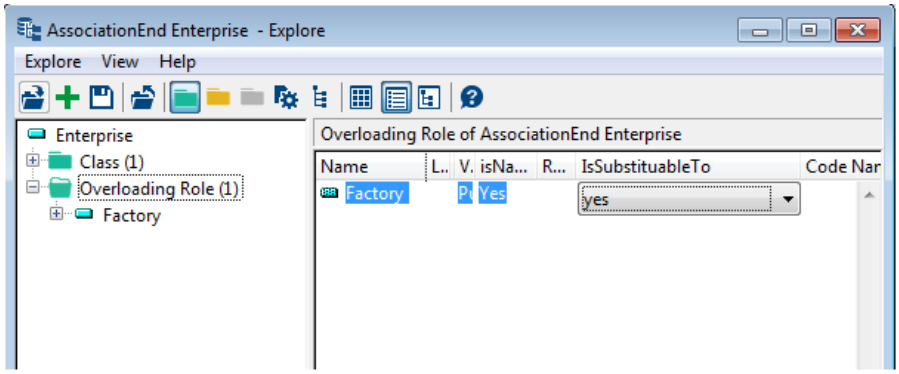
Take for example the "Enterprise" element. To indicate that it is substituted by the "Factory" element:

1. Select the "Enterprise" element in the navigator.
2. In its properties dialog box, select the **Complements** tab.
3. Select "Overloading role" and click the **Connect** button.
4. In the Choose selector dialog box, find the overloading role and click **OK**.



5. Then right-click the "Enterprise" element and select **Explore**.

6. Click "Overloading role" and in the **IsSubstituableTo** column, select "yes".



Other Features Related to Inheritance

When an element inherits, it has all the attributes of the class it inherits. The following questions then arise:

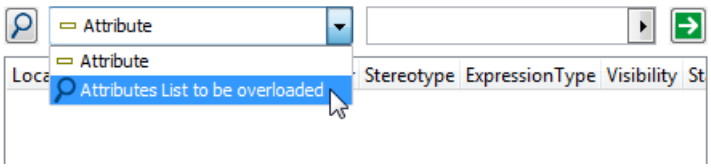
- Is it possible to modify inherited attributes or elements (overloading)?
- Is it possible to use abstract classes?
- What happens when the inherited element is a reference?

Overloading Attributes

The concept of overloading is common in object-oriented programming, but it usually only applies to operation methods. It is a mechanism for defining additional properties on an inherited attribute.

To *overload* an attribute in the schema editor:

1. Select the attribute in the navigator.
2. In its properties dialog box, select the **Complements** tab.
3. Select **Overloaded Attribute** parameter and click the **Connect** button. The Query dialog box appears:
4. Select **List of attributes to be overloaded**.



5. In the results list, select the attribute and click **Connect**.

UML modeling

To overload an attribute in a diagram:

1. Open the properties dialog box for the inherited class.
2. In the **Attribute** tab, select the attribute to be overloaded, then click the **Properties** button.
3. Proceed as described above.

When a class inherits another class, it is possible to modify the inherited attributes.

This overloading (modifying the inherited attributes) is used to define the specific characteristics of the attribute in the child class.

All properties of the attribute that are not redefined in the child class remain identical to those of the parent class.

Result

In the diagram below, the "Web address" attribute is overloaded in the "Web Author" class.

Diagram	Diagram
Web address XML Schema Example::Library::Author.web address	Author +web address[0..1] Web Author +Web address[1]

Restriction: The overloaded attribute must be an attribute inherited from an ancestor class.

The type of the overloading attribute is not defined, as the type is considered to be that of the overloaded attribute (STRING).

Example

The "Author" class has a "Web Address" attribute. This attribute is overloaded in the "Web Author" class. The difference between the original version and the overloaded

version of the attribute is its multiplicity: in the first case the attribute is optional, and in the second case it is required.

XSD

```
<xsd:ComplexType name="Author">
  <xsd:attribute name="lastname" type='xsd:string'
use="required"/>
  <xsd:attribute name="firstname" type='xsd:string'
use="required"/>
  <xsd:attribute name="webaddress" type="xsd:string" />
</xsd:complexType>
<xsd:complexType name="WebAuthor">
   <xsd:simpleContent>
    <xsd:restriction base="Author">
      <xsd:attribute name="lastname" type='xsd:string'
use="required"/>
      <xsd:attribute name="firstname" type='xsd:string'
use="required"/>
      <xsd:attribute name="webaddress" type="xsd:string"
use='required' />
    </xsd:restriction>
  </xsd:simpleContent>
</xsd:complexType>
```

Overloading Elements

Elements can also be overloaded, just as attributes can.

This overloading indicates that element B (overloading element) is substituted for element A (overloaded element) and that the properties of element B (overloading) are defined with priority over A.

> **Restriction: The overloaded element must be an element inherited from an ancestor class.**

To indicate that an element is overloaded in the schema editor:

1. Open the **Properties** dialog box of the overloaded role.
2. Select the **Complements** tab.
3. Select **Overloading role** and click the **Connect** button.
4. In the Choose selector dialog box, find the overloading role and click **Connect**.

UML modeling

To overload elements from a class diagram:

1. Select **View** > **Views and Details** and select the **Schema** check box.
2. In the objects tool bar, click the **Link** button
3. In the diagram, click the overloading role and connect it to the overloaded role.
4. A dialog box will ask you to select the type of link you wish to create. Select **Overloaded element** then click **OK**.

See also "Overloading Attributes", page 64.

You can also overload groups.

Creating Abstract Classes

> *A class can be defined as abstract. This indicates that it cannot be instantiated; there can only be instances of its derived classes.*

To declare a class as abstract:

1. Right-click the class concerned and select **Properties**. The properties dialog box of the type appears.
2. In the **Characteristics** tab, select the **isAbstract** check box and click **Apply**.

UML modeling

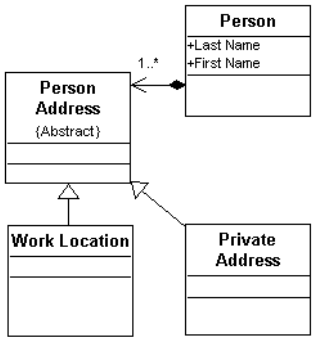
- Display the properties of the class and proceed as described above.

Example

In the example below, the tags defined for the "Person" class have a sub-element called "Address".

It is defined by the "Person Address" class, but this class cannot be instantiated because it is declared as an abstract class.

Therefore only the derived tags (here "Private Address" and "Work Location") can be instantiated.



Example of an instance generated from the schema:

```
<Person LastName="Smith" FirstName="John">
    <PrivateAddress>
    ...
    </PrivateAddress>
    <PrivateAddress>
    ...
    </PrivateAddress>
    <WorkLocation>
    ...
    </WorkLocation>
</Person>
```

Inheritance and References

Function

The "reference" system (see "References", page 42) relies on the addition of an **ID** attribute for each referenced tag, and **IDREF**/**IDREFS** attributes for the referencing tags.

When a tag is referenced from a class using inheritance, the **ID** attribute is propagated to the inherited elements.

☛ *The values of the parameters defining how the attribute is generated are also propagated.*

Example

This case uses the "Paperback" class, which inherits the "Book" class. The attribute identifying the "Book" class is called ISBN. The "Paperback" class inheriting the "Book" class will inherit the "ISBN" attribute.

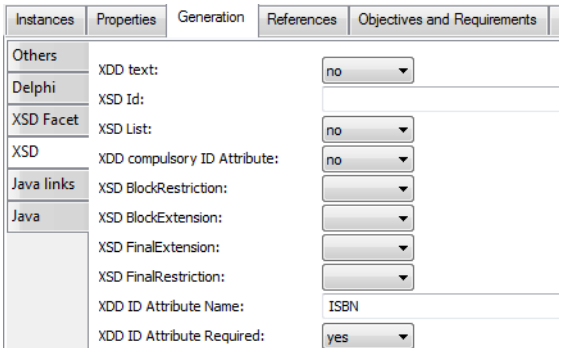
The attribute identifying the "Paperback" class is automatically set to this value.

Thus the **XDD Compulsory ID Attribute**, **XDD ID Attribute Name** and **XDD ID Attribute Required** parameters are set to the values of the inherited class.

The class that references "Book" will also reference "Paperback".

☺ *Of course, it is possible to overload any of these parameters locally.*

Result

Diagram	XML Document
	<Paperback Name="For-WhomTheBellTolls" ISBN="123-456-789"/> <Book Name="A Book " ISBN="789-456-123"/>

Association Classes

Association classes represent an association and specify its characteristics.

To create an association class:

1. Open a **class diagram**, and drag the two associated classes into the diagram.
2. Create a new class.
3. Using the **Link** button, create a link between the class and the association.
4. The association class is linked to the association by a dotted line.

The attributes of association classes are used when building a schema : they correspond to the attributes of the association that are only useful in the context of the element.

When generating schemas, the attributes of association classes migrate to the tag that the associated class defines.

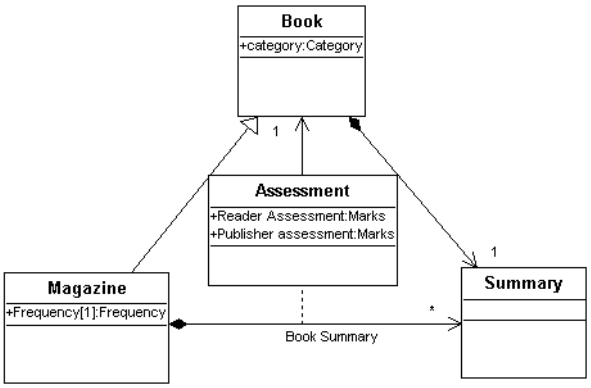
Example

The "Book" class defines a tag containing a sub-element called "Summary".

The tags based on the "Magazine" class also have the summary because the defining class inherits "Book".

However, magazines may contain book summaries (in a book review for example).

In this context, book summaries based on the "Summary" class have the following additional attributes: "Reader Assessment", "Publisher Assessment". These additional attributes are modeled by the "Assessment" association class.



The schemas below show the difference between the "Summary" tag generated for books and the "Book Summary" tag generated for magazines.

XSD

```
<xsd:ComplexType name="Summary">
```

```
    <xsd:attribute name="id" type="xsd:ID"/>
  </xsd:ComplexType>
  <xsd:ComplexType name="BookSummary">
    <xsd:attribute name="id" type="xsd:id"/>
    <xsd:attribute name="Book" type="xsd:idref"
use="required"/>
    <xsd:attribute name="ReaderAssessment" use="required">
      <xsd:enumeration value='VeryGood'/>
      <xsd:enumeration value='Good'/>
      <xsd:enumeration value='Average'/>
      <xsd:enumeration value='NotVeryGood'/>
      <xsd:enumeration value='Bad'/>
     </xsd:attribute>
    <xsd:attribute name="PublisherAssessment"
use="required">
      <xsd:enumeration value='VeryGood'/>
      <xsd:enumeration value='Good'/>
      <xsd:enumeration value='Average'/>
      <xsd:enumeration value='NotVeryGood'/>
      <xsd:enumeration value='Bad'/>
     </xsd:attribute>
  </xsd:ComplexType>
```

Result

Diagram	Diagram	XML Document
Book Summary (Summary) Assessment	Assessment +Reader Assessment:Marks +Publisher assessment:Marks Magazine +Frequency[1]:Frequency Book Summary Summary	<Magazine category="Literature" Frequency="Weekly" BibliographicReference="125" ISBN="1" Author="Jack" <TitleSummary ReaderAssessment = "Good" PublisherAssessment = "Average"></TitleSummary></Magazine>

To view the association class in the XML schema editor:

- click on the **Display types** button and open the association for which the association class is defined.

CLASS STEREOTYPES

You can define one of the following stereotypes for a class:

- Schema group
- Expression
- Enumeration
- Structure (XSD)

Defining a Class Stereotype

To define a stereotype:

1. Open the **Properties** dialog box of the tag (or class) definition.
2. Click the **Characteristics** tab.
3. In the **Stereotype** list box, select "Schema Group", "Enumeration" or "Expression" using the arrow.

Schema Group Stereotype

*A class of the **Schema Group** stereotype represents an ordered group or a subset of a set of elements or attributes.*

The possible orders for a group are:

All/ type elements can appear zero or one times and in any order. "All" groups are only authorized at the highest level of a type and must appear alone. In addition, they can only contain elements.

Sequence: the elements are declared in the order they are defined in the group.

Choice: only one element of the group can be declared.

None: no order is applied.

These ordering options are combined with the values of the multiplicities.

Expression Stereotype

Classes of the **Expression** stereotype are used to define variants of primitive types. They are based on the standard types provided (**Standard::Types::XML::XSD** package).

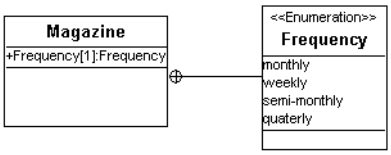
Enumeration Stereotype

Classes of the **Enumeration** stereotype are used to define a set of values. These classes can be used as an attribute type, or even an element type when the schema language allows (such as XSD). Attributes based on such a type can only assume the values defined by the **Enumeration** class.

☛ *It is recommended that **Enumeration** types be declared locally in a class C that defines the attributes based on these types. This*

recommendation only applies when the type is not reused by other attributes external to class C. This avoids proliferation of types in the package and allows reusing the same names for similar concepts.

Example



The "Magazine" class has an attribute called "Frequency". This attribute is based on the "Frequency" type of the "Enumeration" stereotype. The possible values are therefore "monthly", "weekly", "semimonthly", and "quarterly".

As described in the above recommendation, the "Frequency" class is declared locally in the "Magazine" class.

XSD

```
<xsd :simpleType name='FrequencyMagazine'>
<xsd:restriction base='xsd:string'>
  <xsd:enumeration value="monthly"/>
  <xsd:enumeration value="weekly"/>
  <xsd:enumeration value="semimonthly"/>
  <xsd:enumeration value="quarterly"/>
</xsd:restriction>
</xsd:simpleType>
<xsd:ComplexType name="Magazine">
<xsd:complexContent>
     <xsd:extension base='book'>
      <xsd:sequence>
       <xsd:element name="Summary" type="Summary"
minOccurs="1" maxOccurs="1"/>
       <xsd:element name="BookSummary" type="BookSummary"
minOccurs="0"             maxOccurs="*"/>
      </xsd:sequence>
      <xsd:attribute name="Frequency"
type="FrequencyMagazine" use="Required"/>
   </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
```

Structure Stereotype (XSD)

Classes of **Structure** stereotype are used to define unions of types. This stereotype is specific to XSD. See "Union definition", page 50.

CONTROLS

You can check the validity of schemas generated in **MEGA**.

The following points are covered here:

- ✓ "About Checks", page 76
- ✓ "Checking XML Schemas", page 77
- ✓ "UML Checks", page 79

About Checks

☛ *The following paragraphs use UML terminology to describe the checks that are performed. In fact, the check tool is applied to UML concepts.*

The check functionality will find and indicate the anomalies found in the schema. A check can be executed before generating the schema.

To run a check:

1. Right-click the package and select **Check** > **Check Descriptor**.
2. Select one of the checks in the box displayed:
   - **Check Package (XML Document Definition)**: check of compatibility of modeling for generation of XML schemas XML (XSD).
   - **Check Package (UML Semantic)**: check of compatibility of modeling with UML semantics.

CHECKING XML SCHEMAS

Checks are run by object type. This level of checking indicates problems found in the schema modeling.

Checking Associations

An association must be binary.

The control module signals all non-binary associations (ternary, etc.).

The role opposite an aggregation must be navigable.

If elements are described by a binary association in which one role is an aggregation, the opposite role must be navigable (isNavigable set to **Yes** or not specified)

Checking Association Roles

An association role of a primitive type must be named.

When an association role is not named, it automatically assumes the name of the class to which it is attached. For primitive types (**NMTOKENS**, **STRING**), the name of the role must be specified in order to avoid generating elements with reserved names.

Checking Classes

A group can only inherit groups

A class of the **Schema Group** stereotype can only inherit classes of the same stereotype. In this case, inheritance of classes of the **Enumeration** or **Interface** stereotype is prohibited.

A class without a stereotype cannot inherit groups

A class that is not of the **Schema Group** stereotype cannot inherit groups. However, the standard UML stereotypes such as **Boundary**, **Entity**, etc. are not checked as they should not appear in a schema (unless the data model and the document structure are modeled by the same classes).

A class can have only one attribute of type **ID**.

An XML tag can only have one XML attribute of type **ID**.

The constraint in **MEGA** therefore concerns classes that model these tags.

A class must have at most one attribute of type ID.

However, this constraint is made more complex by the automatic creation mode applied to attributes of this type. Attributes inherited by a class that have not been overloaded must also be taken into account.

This check will indicate all attributes of type **ID**, as well as the attribute created automatically when the corresponding option is selected (**XDD compulsory ID Attribute** parameter).

Checking Attributes

Non-optional attributes should not have an initial value

Required attributes (multiplicity of **1** or unspecified) cannot have an initial value because they must always be specified.

The cardinality of an attribute cannot be greater than 1

In the XML language, a tag cannot declare an attribute more than once. Its multiplicity in **MEGA** will therefore be at most 1. The authorized values are: **unspecified**, **0**, **0.0.1** and **1**.

UML CHECKS

The UML checks provided by **MEGA Designer - Development** and **MEGA for UML** correspond to the list defined by **UML**. These include checking for recursion in the inheritance hierarchy, verifying that the attributes of a class or classes owned by a package do not have the same names, etc.

COMPONENTS

The following points are covered here:

- ✓ "Generating a Component", page 82
- ✓ "Component structure", page 83

Generating a Component

A component is a group of objects to be generated.

To create a component:

1. Click the namespace (the package) in the schema and select **New > Component** in the pop-up menu.
2. In the **New component** creation dialog box, enter the **Name** of the component, and select its **Type**, that is its generation target.
3. Click the **Next** button.
4. You can specify the contents of the component (packages, classes, sub-components). It is not essential to do this immediately since you can easily do it later, for example in the navigator or in a component diagram.
5. Click **Next**.
6. Specify the generation folder and the other generation characteristics.
7. Click **Next**.
8. The dialog box that opens enables you to specify the stereotype of the component Click **Finish**.

The component appears in the menu tree of the package.

COMPONENT STRUCTURE

There are two types of component structure:

- the **File** stereotype
- the **Project** stereotype

File stereotype

A **File** component contains the schemas (classes) to be generated, and by default generates all schemas in succession in a single file.

Project stereotype

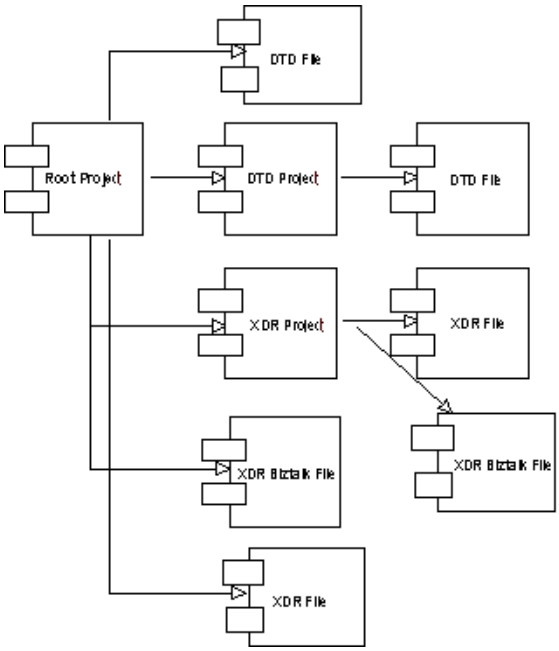
A **Project** component groups sub-projects. It may include:

- classes
- packages
- sub-components (file components or project sub-components).

In a project component, all schemas are by default generated in separate files.

> ☛ *If the project component contains a package, one file will be generated for each **XML Document Definition Root** class in the package.*
>
> ☛ *If the project component contains a sub-component, a file containing all the code for the sub-component will be generated.*

Hierarchy of available components



Root Project

Standard component ensuring the generation of child components of any type.

XSD Project

Generates a file for each element in the component. The possible components are defined in the following sub-list.

- **XSD File**: generates a file in DTD format corresponding to the schema class associated with the component.

COMPLETE EXAMPLES OF XML SCHEMAS

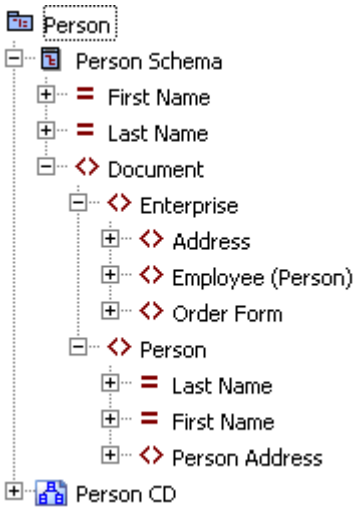
The following examples illustrate the concepts and functions supplied in **MEGA** to construct an XML schema.

Three schemas are presented: "Library", "Person" and "Enterprise".

- ✓ "Person", page 86
- ✓ "Library Example", page 91
- ✓ "Example Enterprise", page 100

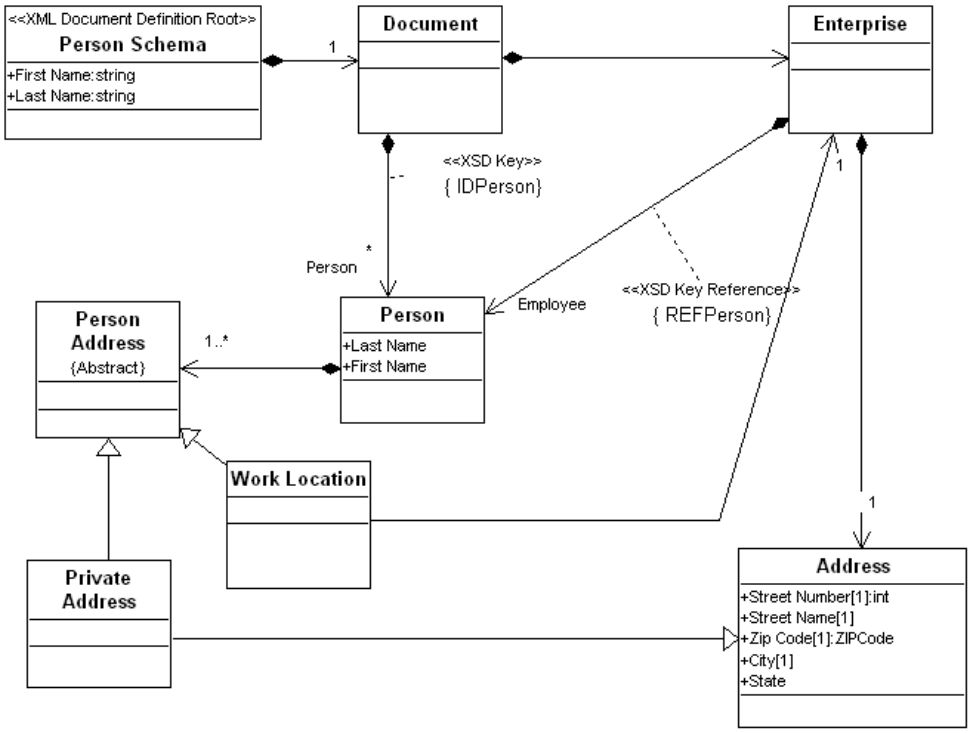
PERSON

Person Schema



Person Schema Diagram

This diagram illustrates the concept of schema in general (class with the **XML Document Definition Root** stereotype), the modeling of XML elements using associations and the use of abstract classes (Person Address).



Person Schema XSD Generation

```
<xsd:schema targetNamespace="Person" xmlns="Person"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:Alias1="urn:schemas-microsoft-com:datatypes"
xmlns:Alias2="Library" xmlns:Alias3="Enterprise">
  <xsd:import namespace="urn:schemas-microsoft-
com:datatypes"/>
  <xsd:import namespace="Library"/>
  <xsd:import namespace="Enterprise" schemaLocation=""/>
  <xsd:element name="Document" type="Document"
minOccurs="1" maxOccurs="1">
  </xsd:element>
  <xsd:attribute name="Name" type="xsd:language"
use="required">
```

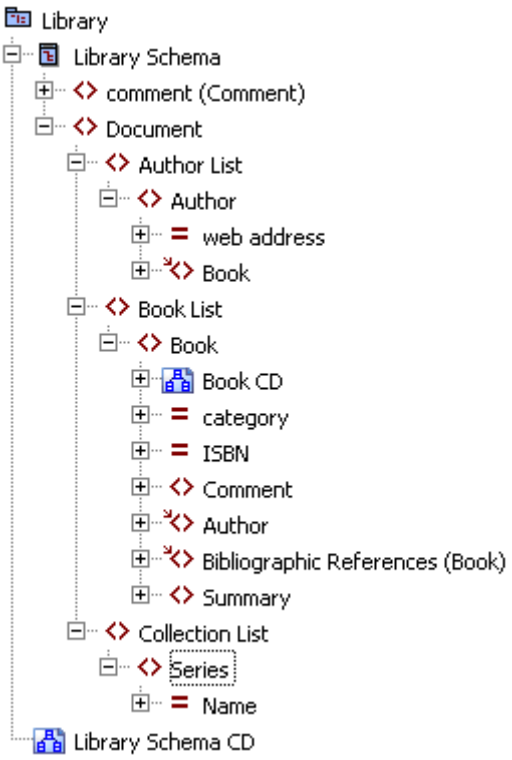
```
  </xsd:attribute>
  <xsd:attribute name="FirstName" type="xsd:language"
use="required">
  </xsd:attribute>
  <xsd:simpleType name="USStateList">
    <xsd:list base="USState">
    </xsd:list>
  </xsd:simpleType>
  <xsd:simpleType name="USState">
    <xsd:restriction base="xsd:language">
      <xsd:enumeration value="PA"/>
      <xsd:enumeration value="NY"/>
      <xsd:enumeration value="CA"/>
      <xsd:enumeration value="LA"/>
      <xsd:enumeration value="AK"/>
    </xsd:restriction>
  </xsd:simpleType>
  <xsd:simpleType name="ZipCode">
    <xsd:union memberTypes="xsd:int USState">
    </xsd:union>
  </xsd:simpleType>
  <xsd:complexType name="Address" abstract="false"
mixed="false">
    <xsd:sequence>
      <xsd:element name="Description" type="Alias2:Comment"
minOccurs="1" maxOccurs="1">
      </xsd:element>
    </xsd:sequence>
    <xsd:attribute name="id" use="required" type="xsd:ID"/>
    <xsd:attribute name="StreetNumber" type="xsd:int"
use="required">
    </xsd:attribute>
    <xsd:attribute name="ZipCode" type="ZipCode"
use="required">
    </xsd:attribute>
    <xsd:attribute name="City" type="xsd:anyType"
use="required">
    </xsd:attribute>
    <xsd:attribute name="StreetName" type="xsd:anyType"
use="required">
    </xsd:attribute>
  </xsd:complexType>
  <xsd:complexType name="PersonAddress" abstract="true"
mixed="false">
    <xsd:attribute name="id" use="required" type="xsd:ID"/>
  </xsd:complexType>
```

```
  <xsd:complexType name="WorkLocation" abstract="false"
mixed="false">
    <xsd:complexContent mixed="false">
      <xsd:extension base="PersonAddress">
        <xsd:attribute name="Enterprise" use="required"
type="xsd:IDREF"/>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
  <xsd:complexType name="PersonalAddress" abstract="false"
mixed="false">
    <xsd:complexContent mixed="false">
      <xsd:extension base="PersonAddress">
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
  <xsd:complexType name="Document" abstract="false"
mixed="false">
    <xsd:sequence>
      <xsd:element name="Enterprise"
type="Alias3:Enterprise" minOccurs="0"
maxOccurs="unbounded">
      </xsd:element>
     <xsd:element name="Person" type="Person" minOccurs="0"
maxOccurs="unbounded">
        <xsd:key name="IDPerson">
          <xsd:selector xpath="Document/Person">
  <xsd:field    xpath="@Name"/>
  <xsd:field    xpath="@FirstName"/>
</xsd:selector>
        </xsd:key>
      </xsd:element>
    </xsd:sequence>
    <xsd:attribute name="id" use="required" type="xsd:ID"/>
  </xsd:complexType>
  <xsd:complexType name="Person" abstract="false"
mixed="false">
    <xsd:sequence>
      <xsd:element name="PersonAddress"
type="PersonAddress" minOccurs="1" maxOccurs="unbounded">
      </xsd:element>
    </xsd:sequence>
    <xsd:attribute name="id" use="required" type="xsd:ID"/>
    <xsd:attribute ref="Name" use="required">
    </xsd:attribute>
    <xsd:attribute ref="FirstName" use="required">
```

```
    </xsd:attribute>
  </xsd:complexType>
</xsd:schema>
```

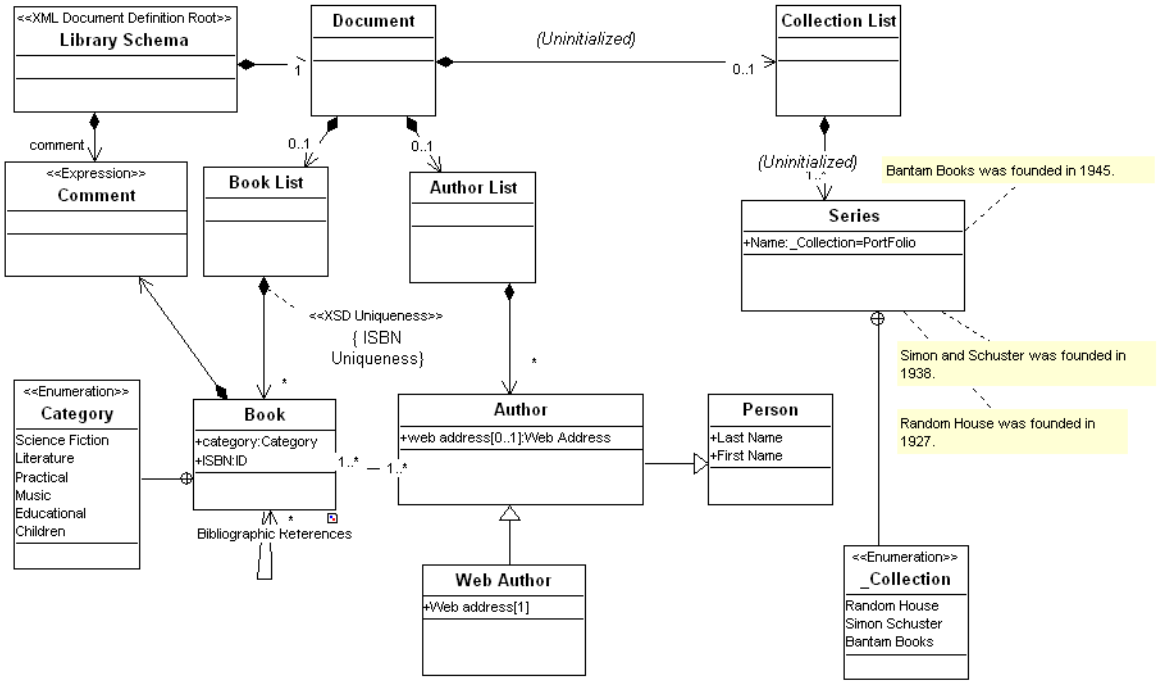
LIBRARY EXAMPLE

Library Schema



Library Diagram

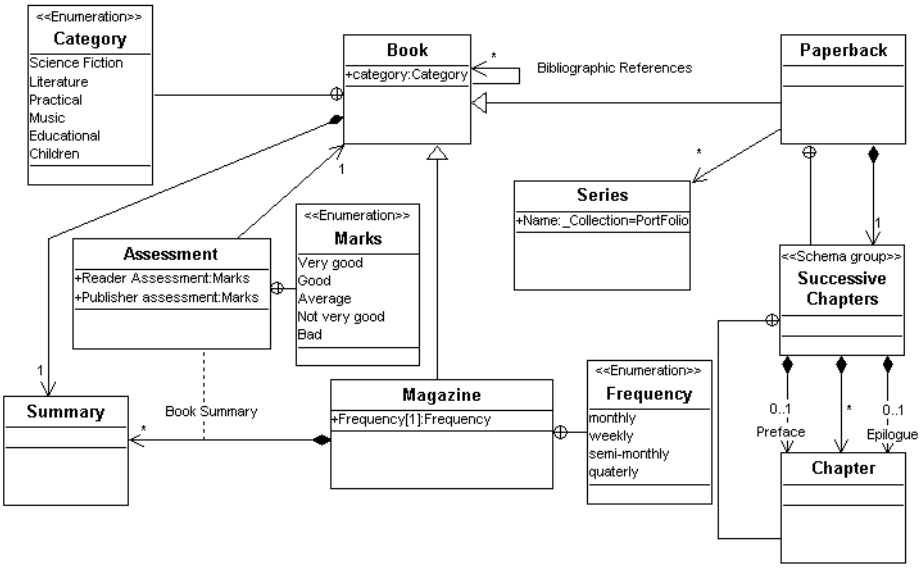
This diagram illustrates bidirectional references (between Book and Author). It also shows attribute overloading (Web Author) and the use of notes.



Book Diagram

This diagram illustrates details related to the "Book" class. You will notice the following class specializations: "Paperback" and "Magazine". The concept of group

is also shown (Successive Chapters). Lastly, an example of an association class is present showing the use of attributes in a specific context (Assessment).



XSD Generation: Library Schema

```
<xsd:schema targetNamespace="Library" xmlns="Library"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:p="Person" xmlns:d="urn:schemas-microsoft-
com:datatypes">
  <xsd:import namespace="Person"/>
  <xsd:import namespace="Person" schemaLocation=""/>
  <xsd:annotation>
    <xsd:documentation>Document root. A document must start
with one of the tags connected to this class.</
xsd:documentation>
  </xsd:annotation>
  <xsd:element name="comment" type="Comment" minOccurs="0"
maxOccurs="unbounded">
  </xsd:element>
  <xsd:element name="Document" type="Document"
minOccurs="1" maxOccurs="1">
  </xsd:element>
  <xsd:simpleType name="Summary">
    <xsd:restriction base="xsd:anyType">
    </xsd:restriction>
  </xsd:simpleType>
```

```
  <xsd:simpleType name="WebAddress">
    <xsd:restriction base="xsd:string">
      <xsd:pattern value="[a-z][A-Z][0-9]+@[a-z][A-Z][0-
9]"/>
    </xsd:restriction>
  </xsd:simpleType>
  <xsd:simpleType name="Comment">
    <xsd:restriction base="xsd:string">
    </xsd:restriction>
  </xsd:simpleType>
  <xsd:complexType name="Magazine" mixed="false">
    <xsd:complexContent mixed="false">
      <xsd:extension base="Book">
        <xsd:sequence>
         <xsd:element name="BookSummary" type="Assessment"
minOccurs="0" maxOccurs="unbounded">
          </xsd:element>
        </xsd:sequence>
        <xsd:attribute name="Frequency" use="required">
          <xsd:simpleType>
            <xsd:restriction base="xsd:anyType">
              <xsd:enumeration value="monthly"/>
              <xsd:enumeration value="quarterly"/>
              <xsd:enumeration value="semimonthly"/>
              <xsd:enumeration value="weekly"/>
            </xsd:restriction>
          </xsd:simpleType>
        </xsd:attribute>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
  <xsd:complexType name="WebAuthor" mixed="false">
    <xsd:simpleContent>
      <xsd:restriction base="Author">
        <xsd:attribute name="id" use="required"
type="xsd:ID"/>
       <xsd:attribute name="WebAddress" type="xsd:anyType"
use="required">
        </xsd:attribute>
        <xsd:attribute ref="Name" use="required">
        </xsd:attribute>
        <xsd:attribute ref="p:FirstName" use="required">
        </xsd:attribute>
   <xsd:attribute name="Book" use="required"
type="xsd:IDREFS"/>
```

```
      </xsd:restriction>
    </xsd:simpleContent>
  </xsd:complexType>
  <xsd:complexType name="Document" abstract="false"
mixed="false">
    <xsd:sequence>
      <xsd:element name="AuthorList" type="AuthorList"
minOccurs="0" maxOccurs="1">
      </xsd:element>
      <xsd:element name="BookList" type="BookList"
minOccurs="0" maxOccurs="1">
      </xsd:element>
      <xsd:element name="CollectionList"
type="CollectionList" minOccurs="0" maxOccurs="1">
      </xsd:element>
    </xsd:sequence>
    <xsd:attribute name="id" use="required" type="xsd:ID"/>
  </xsd:complexType>
  <xsd:complexType name="AuthorList" abstract="false"
mixed="false">
    <xsd:annotation>
      <xsd:documentation>Author list definition.</
xsd:documentation>
    </xsd:annotation>
    <xsd:sequence>
     <xsd:element name="Author" type="Author" minOccurs="0"
maxOccurs="unbounded">
      </xsd:element>
    </xsd:sequence>
    <xsd:attribute name="id" use="required" type="xsd:ID"/>
  </xsd:complexType>
  <xsd:complexType name="Author" abstract="false"
mixed="false">
    <xsd:complexContent mixed="false">
      <xsd:extension base="p:Person">
        <xsd:attribute name="webaddress" type="WebAddress"
use="optional">
        </xsd:attribute>
        <xsd:attribute name="Book" use="required"
type="xsd:IDREFS"/>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
  <xsd:complexType name="BookList" abstract="false"
mixed="false">
    <xsd:sequence>
```

```
      <xsd:element name="Book" type="Book" minOccurs="0"
maxOccurs="unbounded">
        <xsd:unique name="ISBN Uniqueness">
          <xsd:selector xpath="Document/BookList/Book">
  <xsd:field    xpath="@ISBN"/>
</xsd:selector>
        </xsd:unique>
      </xsd:element>
    </xsd:sequence>
    <xsd:attribute name="id" use="required" type="xsd:ID"/>
  </xsd:complexType>
  <xsd:complexType name="Book" abstract="false"
mixed="false">
    <xsd:sequence>
      <xsd:element ref="comment" minOccurs="0"
maxOccurs="unbounded">
      </xsd:element>
      <xsd:element name="Summary" type="Summary"
minOccurs="1" maxOccurs="1">
      </xsd:element>
    </xsd:sequence>
    <xsd :attribute name="category" use="required">
      <xsd:simpleType>
        <xsd:restriction base="xsd:anyType">
          <xsd:enumeration value="Music"/>
          <xsd:enumeration value="Practical"/>
          <xsd:enumeration value="Science Fiction"/>
          <xsd:enumeration value="Literature"/>
          <xsd:enumeration value="Children"/>
          <xsd:enumeration value="Educational"/>
        </xsd:restriction>
      </xsd:simpleType>
    </xsd:attribute>
    <xsd:attribute name="ISBN" type="xsd:ID"
use="required"/>
    </xsd:attribute>
    <xsd:attribute name="BibliographicalReferences"
use="optional" type="xsd:IDREFS"/>
    <xsd:attribute name="Author" use="required"
type="xsd:IDREFS"/>
  </xsd:complexType>
  <xsd:complexType name="Book" mixed="false">
    <xsd:complexContent mixed="false">
      <xsd:extension base="Book">
        <xsd:sequence>
```

```
          <xsd:sequence>
            <xsd:element name="Preface" minOccurs="0"
maxOccurs="1">
              <xsd:simpleType name="Chapter">
                <xsd:restriction base="xsd:anyType">
                </xsd:restriction>
              </xsd:simpleType>
            </xsd:element>
            <xsd:element name="Chapter" minOccurs="0"
maxOccurs="unbounded">
              <xsd:simpleType name="Chapter">
                <xsd:restriction base="xsd:anyType">
                </xsd:restriction>
              </xsd:simpleType>
            </xsd:element>
            <xsd:element name="Epilog" minOccurs="0"
maxOccurs="1">
              <xsd:simpleType name="Chapter">
                <xsd:restriction base="xsd:anyType">
                </xsd:restriction>
              </xsd:simpleType>
            </xsd:element>
          </xsd:sequence>
        </xsd:sequence>
        <xsd:attribute name="Collection" use="optional"
type="xsd:IDREFS"/>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
  <xsd:complexType name="Collection" mixed="false">
    <xsd:annotation>
      <xsd:documentation>The Simon and Schuster collection
was created in 1950</xsd:documentation>
      <xsd:documentation>The Random House collection was
created in 1964</xsd:documentation>
      <xsd:documentation>The Bantam Books collection was
created in 1970</xsd:documentation>
    </xsd:annotation>
    <xsd:attribute name="id" use="required" type="xsd:ID"/>
    <xsd:attribute name="Name" fixed="SimonSchuster"
use="required">
      <xsd:simpleType>
        <xsd:restriction base="xsd:anyType">
          <xsd:enumeration value="RandomHouse"/>
          <xsd:enumeration value="SimonSchuster"/>
          <xsd:enumeration value="BantamBooks"/>
```

```
        </xsd:restriction>
      </xsd:simpleType>
    </xsd:attribute>
  </xsd:complexType>
  <xsd:complexType name="CollectionList" mixed="false">
    <xsd:sequence>
      <xsd:element name="Collection" type="Collection"
minOccurs="1" maxOccurs="unbounded">
      </xsd:element>
    </xsd:sequence>
    <xsd:attribute name="id" use="required" type="xsd:ID"/>
  </xsd:complexType>
  <xsd:complexType name="Assessment" mixed="false">
    <xsd:simpleContent>
      <xsd:extension base="Summary">
   <xsd:attribute name="id" use="required" type="xsd:ID"/>
        <xsd:attribute name="ReaderAssessment"
use="required">
          <xsd:simpleType>
            <xsd:restriction base="xsd:anyType">
              <xsd:enumeration value="Bad"/>
              <xsd:enumeration value="VeryGood"/>
              <xsd:enumeration value="NotVeryGood"/>
              <xsd:enumeration value="Average"/>
              <xsd:enumeration value="Good"/>
            </xsd:restriction>
          </xsd:simpleType>
        </xsd:attribute>
        <xsd:attribute name="PublisherAssessment"
use="required">
          <xsd:simpleType>
            <xsd:restriction base="xsd:anyType">
              <xsd:enumeration value="Bad"/>
              <xsd:enumeration value="VeryGood"/>
              <xsd:enumeration value="NotVeryGood"/>
              <xsd:enumeration value="Average"/>
              <xsd:enumeration value="Good"/>
            </xsd:restriction>
          </xsd:simpleType>
        </xsd:attribute>
        <xsd:attribute name="Book" use="required"
type="xsd:IDREF"/>
      </xsd:extension>
    </xsd:simpleContent>
  </xsd:complexType>
```

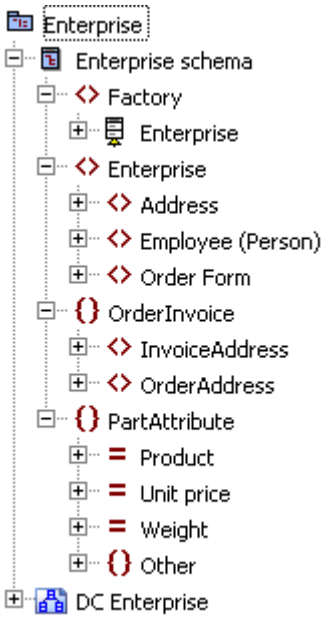
```
</xsd:schema>
```

EXAMPLE ENTERPRISE

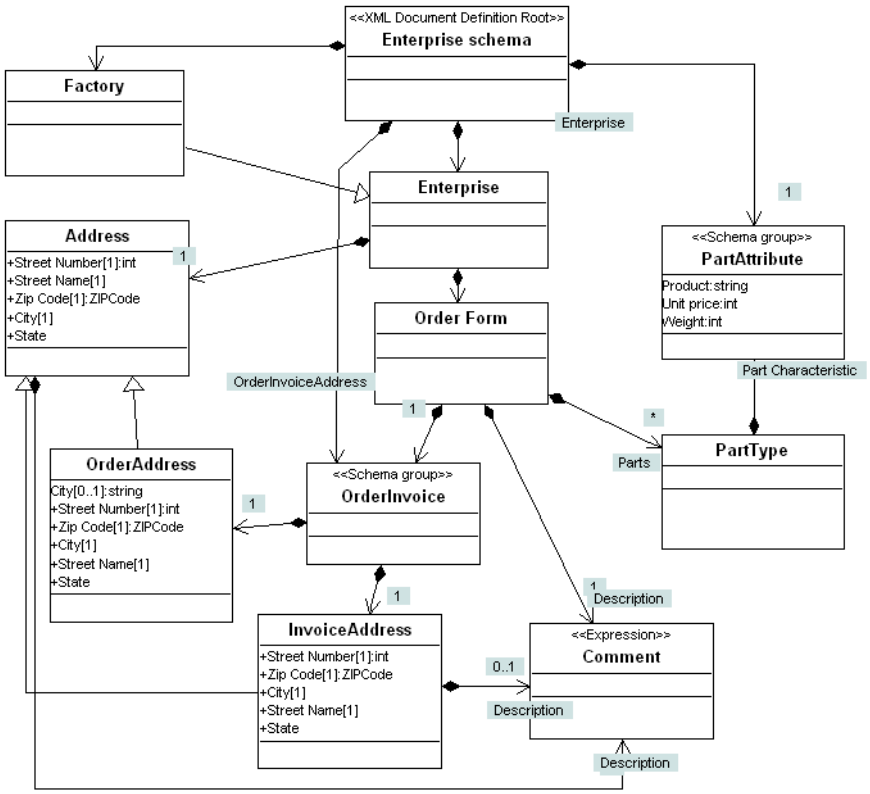
This example illustrates more particularly the functions of XSD, notably:

- Definitions of groups modeled at the schema level
- Substitutions
- Element references
- Inheritance by restriction and extension

Enterprise Schema



Enterprise Diagram



XSD Generation: Enterprise Schema

```
<xsd:schema targetNamespace="Enterprise" xmlns="Enterprise"
xmlns:l="Library" xmlns:Alias1="Person"
xmlns:Alias2="urn:schemas-microsoft-com:datatypes"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:import namespace="Person"/>
  <xsd:import namespace="Library"/>
  <xsd:element name="Factory" type="Factory"
substitutionGroup="Enterprise" minOccurs="0"
maxOccurs="unbounded">
  </xsd:element>
  <xsd:element name="Enterprise" type="Enterprise"
minOccurs="0" maxOccurs="unbounded">
  </xsd:element>
  <xsd:group name="InvoiceOrder">
```

```
    <xsd:sequence>
      <xsd:element name="OrderAddress" type="OrderAddress"
minOccurs="1" maxOccurs="1">
      </xsd:element>
      <xsd:element name="InvoiceAddress"
type="InvoiceAddress" minOccurs="1" maxOccurs="1">
      </xsd:element>
    </xsd:sequence>
  </xsd:group>
  <xsd:attributeGroup name="PartAttribute">
    <xsd:attribute name="Product" type="xsd:string"
use="required">
    </xsd:attribute>
    <xsd:attribute name="UnitPrice" type="xsd:int"
use="required">
    </xsd:attribute>
    <xsd:attribute name="Weight" type="xsd:int"
use="required">
    </xsd:attribute>
    </xsd:anyAttribute>
    </xsd:anyAttribute>
  </xsd:attributeGroup>
  <xsd:complexType name="PartType" mixed="false">
    <xsd:attribute name="id" use="required" type="xsd:ID"/>
  </xsd:complexType>
  <xsd:complexType name="OrderAddress" mixed="false">
    <xsd:simpleContent>
      <xsd:restriction base="Alias1:Address">
   <xsd:sequence>
         <xsd:element name="Description" type="l:Comment"
minOccurs="1" maxOccurs="1">
         </xsd:element>
   </xsd:sequence>

        <xsd:attribute name="id" use="required"
type="xsd:ID"/>
        <xsd:attribute name="City" type="xsd:anyType"
use="optional">
        </xsd:attribute>
        <xsd:attribute name="StreetNumber" type="xsd:int"
use="required">
        </xsd:attribute>
       <xsd:attribute name="ZipCode" type="Alias1:ZipCode"
use="required">
        </xsd:attribute>
```

```
        <xsd:attribute name="StreetName" type="xsd:anyType"
use="required">
        </xsd:attribute>
      </xsd:restriction>
    </xsd:simpleContent>
  </xsd:complexType>
  <xsd:complexType name="InvoiceAddress" mixed="false">
    <xsd:complexContent mixed="false">
      <xsd:extension base="Alias1:Address">
        <xsd:sequence>
          <xsd:element name="Description" type="Comment"
minOccurs="0" maxOccurs="1">
          </xsd:element>
        </xsd:sequence>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
  <xsd:complexType name="OrderForm" mixed="false">
    <xsd:sequence>
      <xsd:element name="Parts" type="PartType"
minOccurs="0" maxOccurs="unbounded">
      </xsd:element>
      <xsd:element name="Description" type="l:Comment"
minOccurs="1" maxOccurs="1">
      </xsd:element>
      <xsd:group ref="InvoiceOrder"/>
    </xsd:sequence>
    <xsd:attribute name="id" use="required" type="xsd:ID"/>
  </xsd:complexType>
  <xsd:complexType name="Enterprise" mixed="false">
    <xsd:sequence>
      <xsd:element name="Address" type="Alias1:Address"
minOccurs="1" maxOccurs="1">
      </xsd:element>
      <xsd:element name="Employee" type="Alias1:Person"
minOccurs="0" maxOccurs="unbounded">
        <xsd:keyref name="REFPerson" refer="IDPerson">
          <xsd:selector xpath="Document/Enterprise/Person">
  <xsd:field    xpath="@Name"/>
  <xsd:field    xpath="@FirstName"/>
</xsd:selector>
        </xsd:keyref>
      </xsd:element>
      <xsd:element name="OrderForm" type="OrderForm"
minOccurs="0" maxOccurs="unbounded">
      </xsd:element>
```

```
    </xsd:sequence>
    <xsd:attribute name="id" use="required" type="xsd:ID"/>
  </xsd:complexType>
  <xsd:complexType name="Factory" mixed="false">
    <xsd:complexContent mixed="false">
      <xsd:extension base="Enterprise">
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
</xsd:schema>
```

XSD GENERATION

The **MEGA for UML** product enables generation of XSD (XML Schema Definition) files from XML schemas or from UML diagrams, either created in **MEGA** or reverse engineered.

This document describes the XSD code generated from an XML schema in **MEGA**.

In order to allow the XML schema to reference **XSD** types, it is necessary to import the corresponding library. See "Presentation of the Schema Editor", page 7.

- ✓ "Generating an XSD Schema", page 106
- ✓ "Namespaces and Schemas", page 107
- ✓ "Type Definitions", page 109
- ✓ "Generating XSD Groups", page 112
- ✓ "Element declaration", page 116
- ✓ "Declaring Attributes", page 121
- ✓ "Model groups", page 124
- ✓ "Notes and Comments", page 127

GENERATING AN XSD SCHEMA

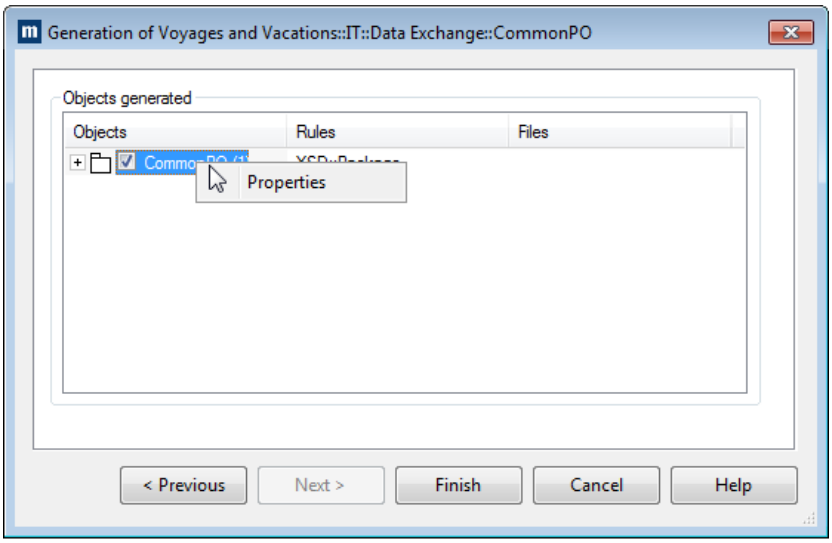
XSD schema generation is accessible from the following objects:

- A UML package of the "XML Document Definition" stereotype
- A UML class of the "XML Document Definition Root" stereotype
- A UML component of the "Project" stereotype
- A UML component of the "File" stereotype

To generate an XSD schema:

1. Right click the object to be generated and select **Generate**.
2. In the dialog box that appears, select **XSD** and click **Next**.
3. In the next dialog box, indicate the generation folder and click **Next**.
   The dialog box displays the objects to be generated.

   You can configure the generation of those objects using their properties dialog box.



4. Click **Finish**.

NAMESPACES AND SCHEMAS

Generating the namespace

A namespace is represented by a package of the "XML Document Definition" stereotype. Packages of this type contain one or more classes of the "XML Document Definition Root" stereotype, and a certain number of classes defining the elements that form the vocabulary of the namespace.

The "XDD urn" generation parameter for the package corresponds to the identifier for the namespace (urn).

It is very important that in the hierarchy of packages that own a class, only one package has the XML Document Definition stereotype. In other words, a namespace cannot own another namespace or be owned by another namespace.

Other packages

You can use packages not interpreted as namespaces for organizational purposes. You can group packages representing namespaces in one or more packages that are not of the "XML Document Definition" stereotype. Similarly, classes defining the vocabulary can be "stored" in packages included in the "namespace" package.

The schema Class

The schemas are represented by classes of the "XML Document Definition Root" stereotype.

The vocabulary contained in the schema is connected to the schema class by the "XML Document Definition Tag" leg, to allow partitioning the namespace. If tag definitions are contained in the "namespace" package without being connected to a schema class, the tag definitions appear in each schema in the namespace.

The root class also serves as a point of departure for declaring global attributes or elements (associated with the attributes of UML classes and roles of UML classes respectively).

You can configure a schema class in the **XSD** tab of its Properties dialog box:

- **XSD Name**: name of the schema class in the context of XSD generation. The name of the generated file containing the schema is calculated from the name of the schema class (or the "XSD Name" parameter if it is present).
- **XSD AttributeFormDefault**: corresponds to the "attributeFormDefault" attribute.
- **XSD BlockExtension**, **XSD BlockRestriction**, and **XSD BlockSubstitution**: used to define the values of the "blockDefault"

attribute. The value of the "blockDefault" attribute is a combination of the "extension", "restriction", and "substitution" values.

- **XSD ElementFormDefault**: corresponds to the "elementFormDefault" attribute.
- **XSD FinalExtension** and **XSD FinalRestriction**: allow defining the values of the "finalDefault" attribute. The value of the "finalDefault" attribute is a combination of the "extension" and "restriction" values.
- **XSD id**: value of the "id" attribute.
- **XSD version**: corresponds to the "version" attribute.
- **XSD xml:lang**: corresponds to the "xml:lang" attribute.
- **specificationLocation**: indicates the location of the schema. It is primarily used for imports and includes.

In the **Alias** tab, you can connect a namespace to the schema class. See "Import and Inclusion of Other Schemas", page 27.

Using Elements External to the Schema

You can use elements external to the schema.

These elements are:

- Global elements
- Global attributes
- "attributeGroups" (definitions of attribute groups)
- Groups of elements (definitions of model groups)
- Simple and complex types

You can also declare other schemas and namespaces that have vocabulary you want to use. See "Import and Inclusion of Other Schemas", page 27.

TYPE DEFINITIONS

In general, UML classes are used to define the various elements in the vocabulary of a namespace, including attributes, elements, model groups, and attribute groups.

Aside from nested classes and associative classes, a class defining a type is generally connected to a schema class by the "XML Document Definition Root" leg. Otherwise, it is considered as belonging globally to the namespace and is therefore generated in all schemas of the namespace.

In the type definitions, XSD offers the ability to define complex types (complexType) and simple types (simpleType).

A class used as a type definition should not be of the "XML Document Definition" stereotype (reserved for schema classes) or of the "Schema group" stereotype (reserved for classes that define groups).

Associative classes are only taken into account in associations used to specify the existence of sub-elements in a type, meaning compositions. In this case, we consider the element to be of the type defined by the associative class and that this is derived from the component class.

Generation Parameters for a Type Class

You can specify the generation parameters for a type in the **XSD** tab of its Properties dialog box:

- **XSD Name**: name of the type in the context of XSD generation. If this is not specified, the name of the class is used.
- **XSD Id**: value of the "id" attribute.
- **XSD BlockExtension** and **XSD BlockRestriction**: are used to determine the value of the "block" attribute. The value of the "block" attribute is a combination of the "extension" and "restriction" values.
- **XSD FinalExtension**, **XSD FinalRestriction**, **XSD FinalList** and **XSD FinalUnion**: allow defining the value of the "final" attribute. This value is a combination of the "restriction", "list", and "union" values if it is a

simple type, and a combination of the "extension" and "restriction" values if it is a complex type.

- **XSD List**: used for simple types, this parameter is used to define "inheritance by list". See "Inheritance by list", page 110.
- **XSD union**: used for simple types, this parameter is used to define "inheritance by union".
- **XDD order**: indicates the ordering constraint on the sub-elements. reference
- **XDD Text**: specifies whether the type contains data other than sub-tags. The result is reflected in the "mixed" attribute.
- **XDD compulsory ID Attribute**: this parameter is used together with **XDD ID Attribute Name** and **XDD ID Attribute Required**. It indicates the presence of an "ID" attribute.
- **XDD ID Attribute Name**: used to indicate the name of the ID attribute generated if **XDD compulsory ID Attribute** is "yes".
- **XDD ID Attribute Required**: used to specify whether the attribute must be present in the document instance.

Type Inheritance

Simple types

Simple types are used to define the data that are contained either in attributes or in tags.

```
<tag attribute="data"/>
<tag>data</tag>
```

A simple type can be derived from another simple type. If a class is derived from another class representing a complex type, the derived class is interpreted as being of a complex type.

☛ *Classes of the expression stereotype are generated as classes deriving from a simple type, and the primitive type is identified by the class connected by the "UML primitive type of expression" leg.??*

Three forms of inheritance are possible for simple types:

Inheritance by restriction

It is possible to create a simple type from another simple type by imposing restrictions (on the size of the data, on the possible values, etc.). See "Restriction definition (facets)", page 53.

Inheritance by list

You can create a type that designates a list of values of a certain type. See ""List" type definition", page 54.

Inheritance by union

You can create types representing a grouping of different types. See "Union definition", page 50.

Complex types

Complex types define the content of tags (declared with the "element" tag) in terms of data, sub-tags, and attributes.

Similarly to simple types, complex types can benefit from inheritance. You can either extend or restrict their content.

A complex type can be derived from a complex type (<complexContent> or <simpleContent> if there is a restriction) or from a simple type (<simpleContent>).

Two forms of inheritance are possible:

Inheritance by extension

You can add sub-tags or attributes to an already existing type.

To extend the content of a complex type:

1. In the class diagram, click the **Generalization** button in the objects toolbar, and connect the class to the class representing the type you want to extend.
   The new type is generated with a <complexContent> or <simpleContent> tag, in which the <elements> or <attributes> are generated inside the <extension> tag.
2. Right-click the generalization and open its **Properties**.
3. Select the **Generation** tab.
4. For **XSD Inherited by restriction**, select "extension".

Inheritance by restriction

You can restrict the content of a complex type. For example, you can change the constraints on its elements or attributes (such as the multiplicity).

To restrict the content of a complex type:

1. Use a generalization to connect the derived class to the class that you want to restrict.
2. Open the properties dialog box of the generalization.
3. Select the **Generation** tab.
4. For **XSD Inherited by restriction**, select "restriction".

In the content of the derived class, you need to add the attributes or create the compositions necessary to redefine the elements for which the constraints have changed.

As for the attributes, connect the redefined attribute to the basic attribute using the "Overloaded attribute" link.

Similarly, a role representing the new use of a sub-element is connected to the basic role by the "overloaded role" leg.

The other elements are not added to the class that will be generated as a restricted type: they will be generated automatically in the new type. The elements of the type are contained in a <restriction> tag, itself contained inside a <complexContent> or <simpleContent> tag.

GENERATING XSD GROUPS

Groups are used to define organizational constraints on the elements contained in a tag type.

According to the XSD specifications, all element declarations must be done within groups.

Generally during generation, the groups are inferred from the classes of the "schemaGroup" stereotype. This type of class is also used to represent model groups of elements and attributes.

In certain cases, the generated groups do not correspond to a class of the "schemaGroup" stereotype, but come from a class representing a type of element. The ordering constraint is specified by the **XDD order** generation parameter, displayed in the Properties dialog box of a class.

The different groups generated are:

all

An "all" content authorizes the elements of the type to appear zero or one times, and in any order. "All" groups are only authorized at the highest level of a type and must appear alone. In addition, they can only contain elements.

> ☛ *The number of times the element appears is indicated by the multiplicitiy declared on the element.*

choice

A "choice" content results in the appearance of only one element among those declared in the group.

sequence

A "sequence" group requires the elements in the group to appear in the order in which they are declared in the group.

any

The "Any" group is a group that may contain any type of element. However, you may restrict the group of elements to a set of namespaces.

You can modify the **XSD Any** parameter in the group properties dialog box. The values available are:

- "any": defines a group of any element type.
- "anyAttribute": defines a group of any attribute type.

The manner in which groups are generated varies as follows:

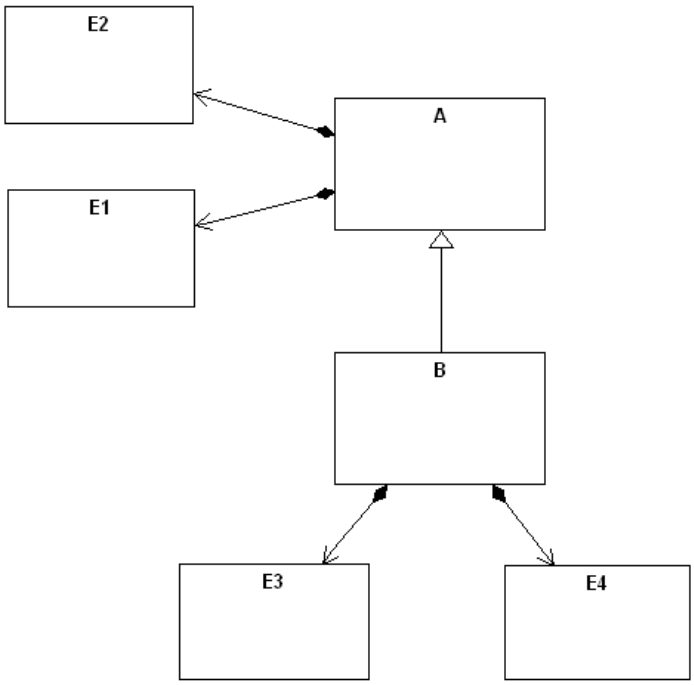
- If the group contains sub-elements, the group is generated following the defined order ("sequence", "choice" or "all").
- If the group does not contain sub-elements, it is generated according to the value ("any" or "anyAttribute") you entered for the **XSD Any**

parameter. Nothing is generated if the **XSD Any** parameter value is not defined.

Inherited Groups

It could be that a type B containing a group inherits from a type A containing a group of the same type. The associated generated XML schema will specifiy the inheritance notion.

However, in the instance document resolved by the schema, the type B element will not contain a group with concatenation of elements of A and B, but two groups: that defined in A then that defined in B.




```
<xsd:complexType name="A" mixed="false">
     <xsd:sequence>
        <xsd:element name="E1" type="E1" minOccurs="0"
maxOccurs="unbounded"/>
        <xsd:element name="E2" type="E2" minOccurs="0"
maxOccurs="unbounded"/>
     </xsd:sequence>
     <xsd:attribute name="id" type="xsd:ID" use="required"/
>
   </xsd:complexType>
   <xsd:complexType name="B" mixed="false">
     <xsd:complexContent mixed="false">
        <xsd:extension base="A">
           <xsd:sequence>
              <xsd:element name="E3" type="E3" minOccurs="0"
maxOccurs="unbounded"/>
              <xsd:element name="E4" type="E4" minOccurs="0"
maxOccurs="unbounded"/>
           </xsd:sequence>
        </xsd:extension>
     </xsd:complexContent>
```

```
</xsd:complexType>
```

To obtain just one generated group, you must define a type B that does not inherit from type A but to which you add elements or attributes of A.

ELEMENT DECLARATION

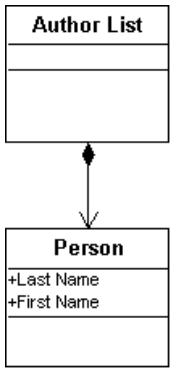
Complex types allow defining tags that can contain other tags. Certain constraints can be used to specify:

- the number of tags of a type that can be contained in it (multiplicity)
- whether the element is abstract
- a value that the tag must be assigned
- etc.

Modeling

Tags contained in a tag type are inferred from the composition associations between the class representing the containing type (composition class) and the classes defining the contained tags (component classes). The declaration characteristics are supported by the association role on the side of the contained tag type.

Example




```
<XSD:complexType name="Listofauthors">
  <XSD:element name="Person" type="Person"/>
</xsd:complexType>
```

From the schema editor, you can specify the following generation parameters on the role:

XSD tab:

- **XSD Name**: name of the tag in the context of XSD generation. If this is not specified, the name of the role is used. If the role has no name, the name of the class connected to the role is used.
- **XSD Id**: identifier for the element tag.
- **XSD BlockExtension**, **XSD BlockRestriction**, and **XSD BlockSubstitution**: used to define the values of the "block" attribute. The value of the "block" attribute is a combination of the "extension", "restriction", and "substitution" values.
- **XSD FinalExtension** and **XSD FinalRestriction**: used to define the values of the "final" attribute. This value is a combination of the "extension" and "restriction" values.
- **XSD Form**: value of the "form" attribute.
- **XSD Value**: specifies the value of the element. This parameter is used in conjuction with the "isChangeable" attribute.
  - If "isChangeable" = "no", a "fixed" attribute is generated. Its value corresponds to the XSD Value parameter.
  - If "Modifiable" = "yes", a "default" attribute that is assigned this value is generated.
- **XSD Abstract**: value of the "abstract" attribute.
- **XSD Nillable**: value of the "nillable" attribute.

Characteristics tab:

- **Multiplicity**: specifies the minimum ("minOccurs" attribute) and maximum ("maxOccurs" attribute) number of times the tag can appear in the containing tag.
- **Editable**: when this attribute is "no", a "fixed" attribute appears in the element tag (if a value is given to the XSD Value parameter).

Keys

Keys implement the concepts of uniqueness (primary key) and of referencing an element.

In an XSD document, keys are used to specify that the elements of a tag (such as an attribute) must have the same value as the elements of another tag in the document instance.

Keys are formally defined in **MEGA** by "Constraints" connected to the class role. The name of the generated key is copied onto that of the constraint. The content of the generated <key> tag is identical to the text contained in the "XSD constraint" attribute of the constraint. This text contains the definition of the key, and must conform with XSD specifications (use of the <selector> and <field> tags). See "XSD: Constraints", page 55.

There are also keys that result in tag generation:

<unique>

This tag is generated from a constraint of the "XSD uniqueness" stereotype. This constraint indicates that there cannot be two tags (containing the key) in which the key elements have the same value. See "Uniqueness in XSD", page 55.

<key>

This key has the same properties as <unique>. In addition, it can be used as a reference for a <keyref> key. The constraint representing this key has the "XSD Key" stereotype. See "Primary Key and Foreign Key in XSD", page 56.

<keyref>

This is a constraint of the "XSD Key Reference" stereotype. See "Primary Key and Foreign Key in XSD", page 56.

Referencing Elements

It is possible to define an element in the schema. This element can then be referenced within a tag definition. The purpose is to avoid redundant definitions. See "Referencing an Element (XML Schema)", page 46.

Substitution

It is possible to declare than an element (called the head element) can be substituted for another in the document instance. This is a "substitution group". The substituted element must be declared globally.

To create the substitution:

1. In the class diagram, create an element of the same type as the head element, or a derived type.
2. Connect the role of the substituting element to the element being replaced. This is an "Overloaded role" link.
3. Open the Properties dialog box for the link.
4. Click the **Generation** tab, then the **XSD** sub-tab.
5. In the **IsSubstituableTo** box, select "yes".

Overloading in an Inheritance by Restriction

When a type is derived by restriction, you can redefine the elements contained in the basic type. For example, you can reduce the multiplicity.

To overload an element:

1. In the class diagram, create a composition between the derived type and the type of the overloaded element.

2. Connect the role representing the overloading element to the one being overloaded. This is an "Overloaded role" link.
3. Open the Properties dialog box for the link.
4. Click the **Generation** tab, then the **XSD** sub-tab.
5. In the **IsSubstituableTo** box, select "no".

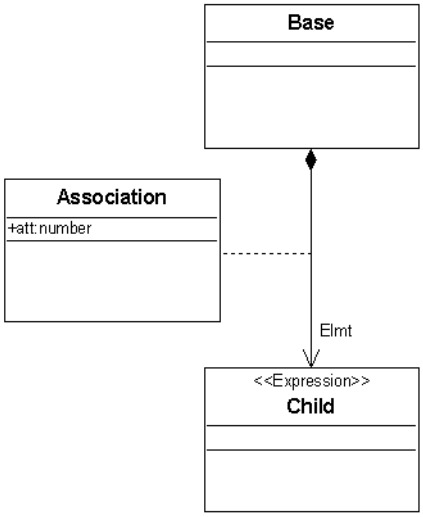
In all cases, the element generated in the derived type will have the same name as the one overloaded.

Associative Classes

Associative classes are only taken into account in compositions. They are used to create elements whose type is an extension of that of the component class. This extended type is only valid in the context of this relation.

In the generated schema, this means creating an element whose type is derived from the component class (as if there existed a generalization between the component class and the associative class).

Example




```
<xsd:complexType name="Base" mixed="false">
    <xsd:sequence>
      <xsd:element name="Elmt" type="Association"
minOccurs="0" maxOccurs="unbounded">
      </xsd:element>
    </xsd:sequence>
  </xsd:complexType>
  <xsd:simpleType name="Child">
    <xsd:restriction base="xsd:NMTOKENS">
    </xsd:restriction>
```

```
  </xsd:simpleType>
  <xsd:complexType name="Association" abstract="false"
mixed="false">
    <xsd:simpleContent>
      <xsd:extension base="Child">
        <xsd:attribute name="att" type="d:number"
use="required">
        </xsd:attribute>
      </xsd:extension>
    </xsd:simpleContent>
  </xsd:complexType>
```

DECLARING ATTRIBUTES

Attribute declarations indicate the presence of attributes in a tag type. Similarly to element declarations, constraints can be applied to attributes:

- optional or required
- default value
- etc.

Modeling attributes

UML attributes for classes correspond to attributes for tag types. To add an attribute declaration to a tag type, you must create an attribute in the class. The attribute type for the tag is inferred from the type of the UML attribute.

The schema editor enables you to parameterize the attribute declarations.

- Select the attribute in the navigator. The attribute properties dialog box is displayed in the left part of the editor.

XSD tab:

- **XSD Name**: name of the attribute in the context of XSD generation. If this parameter is not specified, the attribute name is used for the generation.
- **XSD Id**: used to identify the <attribute> tag.
- **XSD Form**: gives the value of the "form" attribute.
- **XDD UML Attribute as XML element**: specify if UML attributes are generated as XML elements or if they are generated according to their base type. Two options are available:
  - **Determined from the base type**: if the attribute type is simple, an XML attribute is generated. If the attribute type is complex, an XML element is generated.
  - **Enforced as element**: the UML attribute is generated as an XML element, whatever the base type is.

***Characteristics* tab:**

- **Multiplicity**: determines the value of the "use" attribute in the attribute declaration. The multiplicity can have the values "0", "0..1", or "1". The default multiplicity is "1".
  If multiplicity = "0", "use" = "prohibited".
  If multiplicity = "0..1", "use" = "optional".
  If multiplicity = "1", "use" = "required".
- **Editable**: when this attribute is set to "no", a "fixed" attribute appears in the <attribute> tag (if a value is specified for the Initial value parameter).
- **Initial value**: if "Modifiable" is "no", the "fixed" attribute is generated and given the value contained in "Initial value". Otherwise a "default" attribute is generated, its value that of "Initial value".

Attribute References

Just as you can make global element declarations, you can declare attributes at the schema level. These declarations are used in the types as references.

```
<XSD:attribute name="globAttribute" type="globType"/>

<XSD:complexType name="cType">
  <XSD:attribute ref="globAttribute"/>
</xsd:complexType>
```

See "Referencing an Attribute (XML Schema)", page 47.

"ID" Attributes

"ID" attributes provide a way to identify elements uniquely throughout the entire document.

By default, complex types are generated with "ID" attributes. This comes from the fact that the classes have the "XDD compulsory ID Attribute" generation parameter set to "yes". This parameter causes the generation of an "ID" attribute, its name specified by the "XDD ID Attribute Name" parameter, and whether it must be used specified by the "XDD ID Attribute Required" parameter.

> ☛ *You can modify the parameters of all classes in a package by changing the parameters of the package itself.*

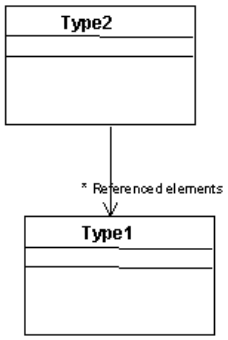
When a UML attribute of the "ID" expression type is present for the class, the parameters specified in the **Generation** tab are ignored. There should not be more than one attribute of type "ID" for a class.

Associations without Composition

"Simple" associations (associations without composition) are not interpreted as element declarations. When an association connects two classes, the navigable roles indicate that the class opposite the navigable role references the adjacent class. This is indicated by creating an attribute of type "IDREF" in the type corresponding to the "referencing" class.

The name of the generated attribute is taken from the name of the role. The multiplicity of the role also determines how the attribute is generated:

- Multiplicity = "0", "use" = "prohibited"
- Multiplicity = "0..1", "use" = "optional"
- Multiplicity = "1", "use" = "required"
- Multiplicity > of 1, the attribute type is "IDREFS"




```
<xsd:complexType name="Type2" mixed="false">
    <xsd:attribute name="id" use="required" type="xsd:ID"/>
   <xsd:attribute name="Referencedelements" use="optional"
type="xsd:IDREFS"/>
  </xsd:complexType>
```

MODEL GROUPS

In XSD generation in **MEGA**, UML classes are used to create content, specifying the sequencing constraints when the content concerns sub-elements or groups of attributes if the class contains attributes.

When the class is used to construct a tag type, this content describes the sub-tags and sub-attributes that can be added to tags of this type.

You can also define content that does not result in tags in the document instance. This content is reusable in tag type definitions. You should use classes of the "schema group" stereotype to do this. See "Generating XSD Groups", page 112.

Creating and Using a model group

The model group classes must be components of the schema class and have the "schema group" stereotype.

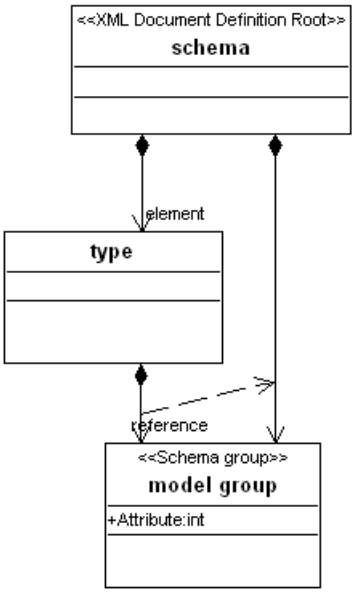
In order to use a model group in a type (or in another group), you must create a composition between the type (or group) and the model group class. The role of the model group (non-composition role) contains information concerning this use (multiplicity for example).

In order to be able to use the model group in the XSD schema, you must reference it.

```
<xsd:group name="modelGroup">
  <xsd:choice>
    <xsd:element name="elmt" type="type1" minOccurs="1"
maxOccurs="unbounded"/>
    <xsd:element name="anotherElmt" type="anotherType"/>
  </xsd:choice>
</xsd:group>
<xsd:complexType name="type2">
  <xsd:sequence>
    <xsd:group ref="modelGroup" minOccurs="1"/>
  </xsd:sequence>
</xsd:complexType>
```

To reference a model group:

- Connect the role of the {type, model group} composition to that of the {schema, model group} composition. This is a link of "Referenced class" type.




```
<?xml version='1.0' encoding='ISO-8859-1'?>
<xsd:schema xmlns="" xmlns:xsd="http://www.w3.org/2001/
XMLSchema">
  <xsd:element name="element" type="type" minOccurs="0"
maxOccurs="unbounded">
  </xsd:element>
  <xsd:attributeGroup name="modelgroupofdattributes">
    <xsd:attribute name="Attrib" type="xsd:int"
use="required">
    </xsd:attribute>
  </xsd:attributeGroup>
  <xsd:complexType name="type" mixed="false">
    <xsd:attribute name="id" use="required" type="xsd:ID"/>
    <xsd:attributeGroup ref="modelgroupofdattributes"/>
  </xsd:complexType>
</xsd:schema>
```

Classes of the "schema group" stereotype are used to model both the model groups of elements and the model groups of attributes. A model group class represents either a group of elements or a group of attributes, but not both.

When a "schema group" class contains one or more attributes, it is considered to be a group of attributes. This results in a model group of attributes for which the tag

in the XSD schema is generated as <attributeGroup> (as are the references in the types).

NOTES AND COMMENTS

One of the various UML concepts used to model schemas in **MEGA** is a text attribute called "comment". It is also possible to connect notes to the objects corresponding to these concepts. These notes can be of type "technical specification" or "documentation".

Comments and notes are generated as <annotation> tags containing <documentation> or <appinfo> tags.

Example

```
<xsd:annotation>
  <xsd:documentation>
    blabla1
  </xsd:documentation>
  <xsd:appinfo>
    blabla2
  </xsd:appinfo>
</xsd:annotation>
```

Comment attributes are generated as <documentation> tags.

Notes that do not have the "technical specification" stereotype are generated as <documentation> tags.

Notes of type "technical specification" are generated as <appinfo> tags.

You can connect an "external reference" to the notes with the "uri Reference" leg. This external reference is used to specify the address of an element supplementing the note. This location is generated in the "source" attribute of the <appinfo> and <documentation> tags.

Objects for which comments and notes are incorporated:

- classes
- Attributes
- roles
- constraints

XSD Reverse Generation

You can import an XSD file into a repository.

This chapter provides an explanation of how XSD schemas are reverse generated and how they are modeled in **MEGA**.

✓ "Importing an XSD Schema in MEGA", page 130
✓ "Modeling the XSD Schema tags", page 131

IMPORTING AN XSD SCHEMA IN MEGA

To import an XSD schema in **MEGA**:

- Click the **File** menu and select **Import** > **XSD**.

In the dialog box that appears, indicate:

- The XSD **Files** to be reverse generated. Generally, the files have the extension .xml or .xsd.
- The **Owner package**, meaning the package into which you want to reverse generate. If you want to create a tree of packages, separate the package names by "::" ". For example, to send information to the "Orders" package, which is inside the "XSD" package, write: "XSD::Orders". The package associated with the namespace for the file being reverse generated will have the targetNamespace of the schema as its name, and will belong to this owner package.
- The **Urn**: once the schemas are selected, if they have the same namespace, the URN is automatically specified with the targetNamespace of the selected schemas. This value cannot be modified.

  ☛ *If the schemas have been retrieved from the Web, their URN is probably an http address (and not the complete path of the schema). If the schemas do not have the same namespace, the URN is grayed out and empty.*
- The **reference package**. Once the reverse generation has been performed, all the referenced schema types that were not found will be stored in this package.
- **Keep MGR file**: when you select this option, the mgr file and the xmgr file from the reverse generation will be saved in the temporary directory on the machine.

MODELING THE XSD SCHEMA TAGS

The XSD schema consists of 20 important tags. Each tag presents a UML modeling.

Schema

The "schema" tag is the root of the XML document. It defines the structure of the XSD schema.

The schema is modeled by a class of the "XML Document Definition Root" stereotype.

Namespace

Its namespace is modeled by a UML package of the "XML Document Definition" stereotype. The name of this package is the value of the "targetNamespace" attribute (the name can be modified). It contains all schemas that define it as a namespace. The namespace is uniquely described by its URN. The URN is modeled by the "XDD Urn" attribute, found in the **XML** generation tab.

A schema may not have a namespace. In this case, the schema is reverse generated into the package of the "XML Document Definition" stereotype which has no URN. If the package does not exist, it is created and given the name "NoTargetNamespace".

Attributes of "schema"

- **Name**: the XSD schema has no name. Therefore the class name becomes the name of the file containing the schema. To generate a name different from that of the class, indicate the new name in the "XSD Name" parameter.
- **AttributFormDefault**: modeled by the **XSD AttributeFormDefault** generation parameter. Its value is either "qualified" or "not qualified".
- **ElementFormDefault**: modeled by the **XSD ElementFormDefault** generation parameter. Its value is either "qualified" or "not qualified".
- **BlockDefault**: modeled by three generation parameters: **XSD BlockExtension**, **XSD BlockRestriction**, and **XSD BlockSubstitution**. Each of these parameters is either "yes" or "no", depending on whether or not the schema defines a blockDefault with a value of extension, restriction, or substitution.
- **FinalDefault**: modeled by the two generation parameters **XSD FinalExtension** and **XSD FinalRestriction**. Each of these parameters is

"yes" or "no" depending on whether or not the schema defines a finalDefault with a value of extension or restriction.

- **TargetNamespace**: modeled by the package containing the schema.
- **Version** : modeled by the **XSD Version** attribute.
- **Xml :lang**: modeled by the **XSD xml :lang** attribute. This parameter has predefined values such as "fr", "de", "en-GB", and "en-US". This list can be extended.
- **SchemaLocation**: this attribute defines the physical location of the schema. It is modeled by the **XSD SpecificationLocation** parameter.

Include - Import

Include

The "Include" tag enables a schema to include types, elements, attributes, and groups from a schema belonging to the same namespace. This provides a way to reuse existing definitions and avoid redundancies. There can thus be a high level of modularity between the schemas used.

The "Include" tag is modeled by the "UML Reference Class-to-Class" link between the including schema and and the included schema.

Attributes of "Include"

- **Id**: modeled by the **XSD Id** link attribute.
- **SchemaLocation**: modeled by the **XSD SpecificationLocation** generation parameter, which is located on the referenced class.
- **Reference**: this is a **MEGA** attribute that can be used to find out whether the viewed schema is included or imported. It is represented by the **XSD Schema Reference** attribute.

Importing

The "Import" tag allows a schema to import types, elements, attributes, and groups from a schema belonging to another namespace. This provides a way to reuse existing definitions and avoid redundancies.

The "Import" tag can be modeled in two ways:

- by the "UML Reference Class-to-Class" link between two classes. In this case, the current schema imports a unique schema.
- by the "UML Reference Class-to-Package" link between a class and a package. The current schema then imports a complete namespace.

Attributes of "Import"

- **Id**: modeled by the **XSD Id** link attribute.
- **SchemaLocation**: modeled by the **XSD SpecificationLocation** generation parameter, which is located on the referenced class. This attribute is only used when importing a schema.
- **Namespace**: modeled by the **XDD Urn** attribute of the package owner of the imported schema or imported package.
- **Reference**: this is a **MEGA** attribute that can be used to find out whether the viewed schema is included or imported. It is represented by the **XSD Schema Reference** attribute. This attribute is only used when importing or including a schema.

SimpleType

The "SimpleType" tag defines a tag definition or XML attribute definition. In other words, it defines the structure of a tag or attribute. This structure can then be reused by multiple tags or attributes. The structure is simple, meaning that it cannot contain any attributes or sub-tags. However, it can represent a list, a union of types, or impose restrictions on already existing simple types.

The "SimpleType" tag is modeled by a UML class.

Attributes of "SimpleType"

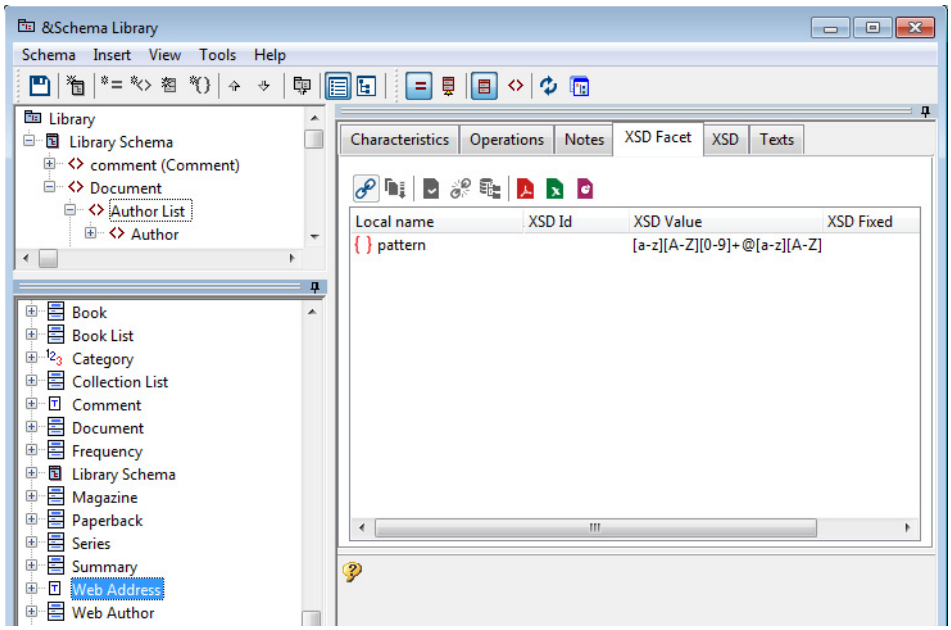
- **Final**: modeled by three generation parameters representing whether certain inheritance types are allowed:
- **XSD Final Restriction** with the tabulated value "yes" or "no"
- **XSD Final List** with the tabulated value "yes" or "no"
- **XSD Final Union** with the tabulated value "yes" or "no"
- **Id**: modeled by the **XSD Id** generation parameter.
- **Name**: modeled by the name of the class. To generate a name different from the name of the class, enter the new name in the **XSD Name** parameter.
- **Type**: this is for modeling which type is reverse generated: This attribute is modeled by the **XSD Type** parameter, which takes the values "simple type" or "complex type". In the present case, it is set to "simple type". It is optional.

Restriction

The primitive type or the internal type for the restriction is modeled by the "UML expression primitive type" link between the simple type and the primitive type. In addition, the simple type is of the "Expression" stereotype. If the type is local, a nested class is created in the simple type.

The facets are modeled by constraints of the "XSD Facet" stereotype on the class (except for an Enumeration).

In the following example, Web Address has a "pattern" facet with the value '[a-z][A-Z][0-9]+@[a-z][A-Z][0-9]'.



For an "enumeration" facet, the simple type is of the "Enumeration" stereotype. The values of the enumeration are stored as literal values.

List

A list is modeled as follows:

- The primitive type or internal type of the list is modeled by a "UML expression primitive type" link between the simple type and the primitive type. In addition, the simple type is of the "Expression" stereotype. If the type is local, a nested class is created in the simple type.
- The list is modeled by the **XSD List** parameter, which is set to "yes" if it is a list or "no" if it is not.

Union

A union is modeled as follows:

- The types can be defined in a list of memberTypes delimited by spaces, and/or locally. They are modeled as attributes of the class associated with the simple type. The name of the attribute is the name of the type. If there are local types, nested classes are created in the simple type.
- Union typing is modeled by initializing the **XSD Union** parameter to "yes" ("no" in the other cases). In addition, the simple type must be of the "Structure" stereotype.

ComplexType

The "ComplexType" tag defines the structure of an XML tag. This structure can then be reused by other tags. The structure is complex, meaning that it may contain attributes and sub-tags. Other types may also be inherited, while restricting them (certain existing elements and attributes are restricted, such as multiplicity) or adding new attributes and elements.

The "ComplexType" tag is modeled by a UML class.

Attributes of "ComplexType"

- **Abstract**: modeled by the **Abstract** attribute of the UML class associated with the type.
- **Block**: modeled by the parameters:
- **XSD Block Extension**: "yes" if preventing substitution by a type inherited by extension in the document instance, or "no" otherwise.
- **XSD Block Restriction**: "yes" if preventing substitution by a type inherited by restriction in the document instance, or "no" otherwise.
- **Final**: modeled by the parameters:
- **XSD Final Extension**: "yes" if preventing inheritance by extension of the complexType, or "no" otherwise.
- **XSD Final Restriction**: "yes" if preventing inheritance by restriction of the complexType, or "no" otherwise.
- **Id**: modeled by the **XSD ID** parameter.
- **Mixed**: modeled by the **XDD Text** parameter. If the value of the attribute is "yes", the XDD Text parameter is set to "yes". Otherwise, it is "no".
- **Name**: defines the name of the type. To generate a name different from that of the class, indicate the new name in the **XSD Name** parameter.
- **Type**: this is for modeling which type is reverse generated: This attribute is modeled by the **XSD Type** parameter, which can have the value "simple type" or "complex type". In the present case, its value is "complex type". It is optional.

SimpleContent

The modeling depends on the type of inheritance:

If the type restricts a primitive type, the following is created:

- A generalization between the two types. Then the **XSD Inherited By Restriction** generalization attribute is initialized to "yes".
- If the restriction has facets, constraints are created as they were for simple types, and the facet values are associated with them.
- If the restriction also restricts attributes of the primitive type, the overloading attribute is created, and the overloading attribute is connected to the overloaded attributed by the "overloaded attribute" leg.
- If the restriction restricts a group of attributes of the primitive type, the overloading group of attributes is created, and the overloading group is connected to the overloaded group by the "overloaded role" leg.

If the type extends a primitive type, the following is created:

- A generalization between the two types. Then the **XSD Inherited By Restriction** generalization attribute is initialized to "no".
- The new attributes and new groups are created.

ComplexContent

The modeling depends on the type of inheritance:

If the type restricts a primitive type, the following is created:

- A generalization between the two types. Then the **XSD Inherited By Restriction** generalization attribute is initialized to "yes".
- If the restriction restricts elements of the primitive type, the overloading element is created, and the overloading element is connected to the overloaded element by the "overloaded role" leg.
- If the restriction restricts attributes of the primitive type, the overloading attribute is created, and the overloading attribute is connected to the overloaded attribute by the "overloaded attribute" leg.
- If the restriction restricts a group of attributes or elements of the primitive type, the overloading group is created, and the overloading group is connected to the overloaded group by the "overloaded role" leg.

If the type extends a primitive type, the following is created:

- A generalization between the two types. Then the **XSD Inherited By Restriction** generalization attribute is initialized to "no".
- The new attributes, new elements, and new groups are created.

Element

The "Element" tag is a declaration of a simple type or a complex type. It represents the instance tag for the document. It may contain attributes, sub-tags, and/or text.

An element often indicates a relation between two classes. This relation is represented by a UML composition. The element is modeled by the role of the class located on the side of the class associated with its type.

Attributes of "Element"

- **Abstract**: modeled by the "XSD Abstract" generation parameter. If the element is abstract, select "yes". Otherwise select "no".
- **Block**: represented by three generation parameters: **XSD Block Extension**, **XSD Block Restriction**, and **XSD Block Substitution**. Depending on the value of Block, the generation parameters are set to "yes" or "no".
- **Default**: modeled by the **XSD Value** generation parameter, which contains the default value for the element, and by the **IsChangeable** attribute, which should not be specified.
- **Final**: represented by the **XSD Final Extension** and **XSD Final Restriction** generation parameters. Depending on the value of Final, the generation parameters are set to "yes" or "no".
- **Fixed**: modeled by the **XSD Value** generation parameter, which contains the default value for the element, and by the **IsChangeable** attribute, which should be initialized to "frozen".
- **Form**: represented by the **XSD Form** generation parameter. If the element is qualified, the parameter is set to "Qualified". Otherwise, it is set to "Unqualified".
- **MaxOccurs**: represented by the maximum multiplicity of the class role.
- **MinOccurs**: represented by the minimum multiplicity of the class role.
- **Name**: modeled by the name of the role.
- **Nillable**: modeled by the **XSD Nillable** generation parameter. If the attribute is nil, the parameter value is "yes". Otherwise, it is "no".
- **Ref**: modeled by the reference link between class roles. This is the "Referenced Role/Referencing Role" link.
- **SubstitutionGroup**: modeled by the **Overloaded role** link. This link is already used to overload elements by restriction. The **IsSubstituableTo** link attribute is "yes" if the element is substituted for another element, and is "no" if the element overloads another element.
- **Type**: modeled by the class with which the class role is associated.

Child

ComplexType

This is a complexType internal to the element. It is modeled by a class. See "ComplexType", page 135.

The local definition of the type is represented by the "UML Nested Class" link. The class is therefore contained in the associated class.

SimpleType

This is a simpleType internal to the element. It is modeled by a class. See "SimpleType", page 133.

The local definition of the type is represented by the "UML Nested Class" link. The class is therefore contained in the associated class.

Unique - Key - KeyRef

Unique

The "Unique" tag defines a uniqueness constraint on an element. The uniqueness constraint generally applies to the value of an attribute or to the text contained in a tag.

Unique: modeled by a constraint of the "XSD Uniqueness" stereotype, and by a constraint type that is set to "unique". This constraint is connected to the class role (element) to be constrained.

Attributes of "Unique"

- **Id**: this attribute is not modeled.
- **Name**: modeled by the name of the constraint.

Selector - Field

The "Selector" tag defines the path for accessing the constrained fields.

"Field" defines the constrained attributes or elements. These attributes or elements are accessed by following the path defined by "Selector".

These tags are modeled by the **XSD Constraint** text field for the constraint.

Annotation

Refer to "Annotation (Documentation - AppInfo)", page 144.

Key Definition

The "Key" tag defines a uniqueness constraint on an element. It has the same role as "Unique", but in addition, you can reference it as a foreign key. The uniqueness constraint generally applies to the value of an attribute or to the text contained in a tag.

"Key" is modeled by a constraint of the "XSD Key" stereotype. The constraint has no associated type. This constraint is connected to the class role (element) to be constrained.

Attributes of "Key"

- **Id**: this attribute is not modeled.
- **Name**: modeled by the name of the constraint.

Selector - Field

These tags are modeled by the "XSD Constraint" text field of the constraint.

KeyRef Definition

The "KeyRef" tag defines a constraint on an element that takes its values from elements subject to another "Key" constraint.

"KeyRef" is modeled by a constraint of the "XSD KeyReference" stereotype. The constraint has no associated type. This constraint is connected to the class role (element) to be constrained.

Attributes of "Keyref"

- **Id**: this attribute is not modeled.
- **Name**: modeled by the name of the constraint.
- **Refer**: modeled by the "Constraint" link between the referencing constraint and the referenced constraint.

Selector - Field

These tags are modeled by the "XSD Constraint" text field of the constraint.

Group

The "Group" tag defines a group of elements in the schema. The goal is to be able to reuse this group in different schemas through references. This avoids redundancies in the element group definitions.

The group is defined by a class of the "schema Group" stereotype. The group declaration is represented by a UML composition. The group occurrence is modeled by the class role on the side of the class of the "schema Group" stereotype. This functions similarly to the way it does for elements.

Attributes of "Group"

- **Name**: modeled by the name of the class role and the name of the class: they both have the same name.
- **ID**: modeled by the **XSD ID** generation parameter for the class, if the group is defined at the schema level. However, if this is a reference, the Id attribute for the class cannot be overloaded. It is then modeled by the **XSD ID** generation parameter defined in the generation tab for the class role.
- **MaxOccurs**: modeled by the maximum multiplicity on the class role.
- **MinOccurs**: modeled by the minimum multiplicity on the class role.
- **Ref**: modeled by the reference link between class roles. This is the "Referenced Role/Referencing Role" link.

Child

A group may have as a child:

- "all", page 140
- "Sequence", page 140
- "Choice", page 141
- "Annotation (Documentation - AppInfo)", page 144

all

The "All" tag defines a group of unordered elements that may appear 0 or one time.

"All" is modeled by a class of the "schema Group" stereotype, in its definition. The declaration of this group is represented by a UML composition. The occurrence of the group is represented by the role of the UML class on the side of the Group class.

The **XDD Order** generation parameter for the class is set to "none".

If the group has no siblings, the name of the class is "All". Otherwise it is set to "All" followed by an index indicating the placement in the group.

If All has no siblings, if its default multicplicity is 1, and if its parent is a type or a model group, the All group is factored to the parent. It is the parent that bears the **XDD Order** parameter.

Attributes of "All"

- **Id**: modeled by the **XSD Id** generation parameter for the class.
- **MaxOccurs**: modeled by the maximum multiplicity of the role for the group class.
- **MinOccurs**: modeled by the minimum multiplicity of the role for the group class.

Child

"All" may have as a child:

- "Annotation (Documentation - AppInfo)", page 144
- "Element", page 136

Sequence

The "Sequence" tag defines a group of elements and/or sub-groups ordered sequentially.

"Sequence" is modeled by a class of the "schema Group" stereotype, in its definition. The declaration of this group is represented by a UML composition, and the group occurrence is represented by the UML class role on the side of the Group class.

The **XDD Order** generation parameter for the class is set to "sequence".

If the group has no siblings, the name of the class is "Sequence". Otherwise, it is "Sequence" followed by an index indicating the placement in the group.

If Sequence has no siblings, if its default multiplicity is 1, and if its parent is a model group or type, the Sequence group is factored up to the parent. It is the parent that bears the **XDD Order** parameter.

Attributes of "Sequence"

- **Id**: modeled by the **XSD Id** generation parameter for the class.
- **MaxOccurs**: modeled by the maximum multiplicity of the role for the group class.
- **MinOccurs**: modeled by the minimum multiplicity of the role for the group class.

Child

"Sequence" may have as a child:

- "Element", page 136
- "Group", page 139
- "Choice", page 141
- "Sequence", page 140
- "Any", page 141
- "Annotation (Documentation - AppInfo)", page 144

Choice

The "Choice" tag defines a group of elements and/or sub-groups. In the document instance, only one of these elements or groups should appear.

"Choice" is modeled by a class of the "schema Group" stereotype, in its definition. The declaration of this group is represented by a UML composition, and the group occurrence is represented by the UML class role on the side of the Group class.

The **XDD Order** generation parameter for the class is set to "choice".

If the group has no siblings, the name of the class is "Choice". Otherwise, it is "Choice" followed by an index indicating the placement in the group.

If Choice has no siblings, its default multiplicity is 1, and if its parent is a model group or type, the Choice group is factored to the parent. It is the parent that bears the **XDD Order** parameter.

Attributes of "Choice"

- **Id**: modeled by the XSD Id generation parameter for the class.
- **MaxOccurs**: modeled by the maximum multiplicity of the role for the group class.
- **MinOccurs**: modeled by the minimum multiplicity of the role for the group class.

Any

The "Any" tag defines a group that may contain any type of element. However, you may restrict the group of elements to a set of namespaces.

"Any" is modeled by a class of the "schema Group" stereotype, in its definition. The declaration of this group is represented by a UML composition, and the group occurrence is represented by the UML class role on the side of the Group class.

The **XDD Order** generation parameter for the class is set to "none". The **XSD Any** generation parameter is set to "any".

If the group has no siblings, the name of the class is "Any". Otherwise, it is "Any" followed by an index indicating the placement in the group.

Attributes of "Any"

- **Id**: modeled by the **XSD Id** generation parameter for the class.
- **MaxOccurs**: modeled by the maximum multiplicity of the role for the group class.
- **MinOccurs**: modeled by the minimum multiplicity of the role for the group class.
- **Namespace**: modeled by the **XSD Namespace** generation parameter. It takes a tabulated value from the following list:
  - ##any: any element is suitable
  - ##other: any element not belonging to the current namespace
  - ##targetNamespace: any element belonging to the current namespace
  - Uri list: any attribute belonging to the list of namespaces
  - ##Local: any element local to a type.
- **processContents**: modeled by the **XSD processContents** generation parameter. It takes a tabulated value from the following list:
  - lax: if the element has a defined, known type, the element must be valid in comparison to its definition. Otherwise, it must simply be well-formed.
  - skip: no constraint is applied. The element must simply be well-formed.
  - strict: the element must be validated once its type is identified in the document instance by a defined type or by a type referenced by the xsi :type attribute.

Attribute

The "Attribute" tag defines an attribute for an XML tag. It is based on a simple type.

The attribute is modeled by a UML attribute of the class that declares it. Thus, if the attribute is defined at the schema level, the class that bears the attribute is the Schema class. If a type is concerned, the attribute will be on the class for the type.

Attributes of "Attribute"

- **Default**: modeled by the **Initial value** and **Modifiable** attributes. **Initial value** shows the default value, and **Modifiable** is set to "yes".
- **Fixed**: modeled by the **Initial value** and **Modifiable** attributes. **Initial value** shows the value of Fixed, and **Modifiable** is set to "no".
- **Form**: modeled by the **XSD Form** generation parameter. This attribute is a tabulated value, and is either "qualified" or "unqualified".
- **Id**: modeled by the **XSD Id** generation parameter.
- **Name**: defines the name of the attribute. The **XSD Name** attribute is also defined, which is the value used during generation. You can therefore change the attribute name without changing the representation of the schema.
- **Ref**: modeled by the "Referenced Attribute/Referencing Attribute" link. This references an attribute defined at the schema level.
- **Type**: modeled by the type of the attribute.
- **Use**: modeled by the multiplicity of the attribute.
  - Optional: the multiplicity of the attribute is 0..1.
  - Required: the multiplicity of the attribute is 1.
  - Prohibited: the multiplicity of the attribute is 0.

AttributeGroup

The "AttributeGroup" tag defines a group of attributes. It is declared at the schema level, then referenced from the types. This avoids redundant definitions.

The attribute group is defined by a class of the "schema Group" stereotype. The declaration of this group is represented by a UML composition. The group occurrence is modeled by the class role on the side of the class of the "schema Group" stereotype.

Attributes of "AttributeGroup"

- **Name**: modeled by the name of the class role and the name of the class: they both have the same name.
- **ID**: modeled by the **XSD ID** generation parameter for the class, if the group is defined at the schema level. If it is a reference, you cannot overload the Id attribute for the class, so it is modeled by the **XSD ID** generation parameter defined in the generation tab for the class role.
- **Ref**: modeled by the reference link between class roles. This is the "Referenced Role/Referencing Role" link.

AnyAttribute

The "AnyAttribute" tag defines a group that may contain any type of attribute. However, you can restrict the group of attributes to a set of attributes belonging to certain namespaces.

"AnyAttribute" is modeled by a class of the "schema Group" stereotype, in its definition. The declaration of this group is represented by a UML composition, and the group occurrence is represented by the UML class role on the side of the Group class.

The **XDD Order** generation parameter for the class is set to "none".

The **XSD Any** generation parameter is set to "anyAttribute".

If the group has no siblings, the name of the class is "AnyAttribute" . Otherwise, it is "AnyAttribute" followed by an index indicating the placement in the group.

Attributes of AnyAttribute

- **Id**: modeled by the **XSD Id** generation parameter for the class.
- **Namespace**: modeled by the **XSD Namespace** generation parameter. It takes a tabulated value from the following list:
  - ##any: any attribute is suitable
  - ##other: any attribute not belonging to the current namespace
  - Uri list: any attribute belonging to the list of namespaces
  - ##targetNamespace: any attribute belonging to the current namespace
  - ##Local: any element local to a type.
- **processContents**: modeled by the **XSD processContents** generation parameter. It takes a tabulated value from the following list:
  - lax: if the element has a defined, known type, the element must be valid in comparison to its definition. Otherwise, it must simply be well-formed.
  - skip: no constraint is applied. The element must simply be well-formed.
  - strict: the element must be validated once its type is identified in the document instance by a defined type or by a type referenced by the xsi :type attribute.

Annotation (Documentation - AppInfo)

The "Annotation" tag is used to associate a descriptor with an XSD concept. The "Documentation" tag provides general information about the concept. The "AppInfo" tag provides technical information.

There are two cases to be modeled:

Only one annotation on the concept

If you have at most one documentation and one appInfo without an associated source, the Documentation tag is modeled by the **Comment** attribute found on all concepts used. The AppInfo tag is modeled by the **XSD AppInfo** text in the XSD generation tab.

Multiple annotations or multiple documentations and appInfos or associated sources

In this case, the Documentation and AppInfo tags are modeled by a Note. For the Documentation tag, the note is of the "XML Documentation" stereotype. For the AppInfo tag, the note is of the "XML Application" stereotype.

Attributes

- **Id** : not modeled.
- **Source**: modeled by an external reference associated with the note representing "Documentation" or "AppInfo".

MEGA DESIGNER - XML SCHEMASGLOSSARY

alias

An alias is a short name given by the user to a URN (the identifier for a namespace).

XML attribute

The XML attributes of a tag are pairs of strings in the format name-value. The second string is contained within quotes. XML attributes serve to characterize the tag. Example: <person name= 'Smith'/>

xml tag

The XML tag is text delimited by the < and > symbols. This text can be recognized by its name (a set of characters containing no spaces) and its attributes. Example: <person>

tag definition

Tag definition is the concept that identifies the names of its attributes, the multiplicity of its attributes, and sub-tags contained in the tag. This concept defines a family of tags (we will use UML classes to define tags). Instances of these tags are found in XML documents. Tag definition should be distinguished from tag.

dependency

A dependency is a one-way relation between elements. The dependency relationship connects elements within the same diagram.

XML element

An element represents a tag in the schema editor The XML elements of a tag A are defined by the set of tags potentially included in A.

namespace The namespace is a set of predefined terms (set of tag definitions, elements and attributes of a document). Inside this space, two tags are not allowed to have the same name.

group A group is a set of elements or attributes. It is usefule to constitute groups to order the elements they contain.

inheritance Inheritance is a link between two classes, by which the child class inherits all the attributes and elements of the parent class.

base References are used to create a relationship between two tags without having to include the referenced tag. For example, if a document contains a list of persons, you can refer to the persons declared in the list without having to include the actual declaration. This eliminates redundancies and facilitates document maintenance.

overloading The concept of overloading is common in object-oriented programming, but it usually only applies to operation methods. It is a mechanism for defining additional properties on an inherited attribute.

urn The URN is the technical identifier for a namespace. It uniquely describes the namespace.

XML XML stands for eXtensible Markup Language. This is a language for describing and exchanging structured documents. The first version of XML was adopted by the W3C in January 1998. Users can add new tags to XML. This is why the language is said to be extensible. The ability to add extensions to the language is the key feature of XML.