YAOUNDE UNIVERSITY I
FACULTY OF SCIENCE
Department of Computer Sciences

THE UML LANGUAGE (UNIFIED MODELING LANGUAGE)

Introduction

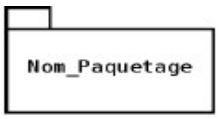
The evolution of programming techniques has always been driven by the need to design and maintain increasingly complex applications. Programming by punch cards, for example, gave way to more advanced techniques, such as assembler (1947); then there were more advanced languages such as Fortran where, until then, programming techniques were based on conditional and unconditional branching (goto), making important programs extremely difficult to develop, master and maintain.
Structured programming (Pascal, C, ...) was then born and allowed the development and maintenance of ever more ambitious applications. Algorithms were no longer sufficient on their own, at the end of the
In the 1970s, software engineering came to place methodology at the heart of software development.
Methods such as Merise (1978) then imposed themselves. The size of applications continued to grow,
Structured programming also met its limits, giving way to object-oriented programming. Object technology is therefore the ultimate consequence of modularization, dictated by the mastery of the design and maintenance of increasingly complex applications. This new
programming technique required the design of new modeling methods. UML (Unified Modeling Language) was born from the merger of the three
Methods that were essential in the field of objective modeling: OMT, Booch and OOSE.
Major industrial players (IBM, Microsoft, Oracle, DEC, HP, Rational, Unisys, etc.) joined the effort and proposed UML 1.0 to the OMG (Object Management Group), which accepted it in November.
1997 in its version 1.1. The current version of UML is UML 2.1.2 which is more than ever necessary in as a standardized modeling language for software modeling.
This document is the support of the UML course given to the Master's level students of the Department of Computer Science of the University of Yaounde I.

1. General notions of the UML language

We present here some notions used in object modeling with UML.

1.1 Package

A package is a grouping of model elements. It allows to organize model elements in groups. It can contain any type of model element: classes, use cases, interfaces, etc. and even nested packages (hierarchical decomposition). A package is represented as a folder with its name written inside or in the tab. The elements contained in a package must represent a highly coherent whole.



1.2 Namespace

Namespaces are packages, binders, etc. An element can be uniquely named by its qualified name, which is the series of names of packages or other namespaces from the root to the element in question. In a qualified name, each namespace is separated by a colon (::). For example, if a package B is included in package A and contains a class X, one must write A::B::X to be able to use the class X outside the context of package B.

1.3 Stereotype

A stereotype is an annotation applied to a model element. It has no formal definition, but allows to better characterize varieties of the same concept. It therefore allows the language to be adapted to particular situations. It is represented by a string of characters between quotation marks (" ") in, or near, the symbol of the basic model element. UML uses rectangles to represent classes, use cases or actors. However, the notation is not ambiguous thanks to the presence of the "use case" stereotype.

UML diagrams

UML is not a method, but a graphical language that allows to represent and communicate the various aspects of an information system. To the graphics are of course associated texts that explain their content. UML is therefore a metalanguage because it provides the elements to build the model that will be the language of the project.

It is impossible to give a complete graphical representation of a software program, or any other complex system, just as it is impossible to fully represent a statue (three-dimensional) by photographs (two-dimensional). But it is possible to give on such a system of partial views, each one analogous to a photograph of a statue, and whose conjunction will give an idea that can be used in practice without the risk of serious error. UML 2.0 includes thirteen types of diagrams representing as many distinct views to represent particular concepts of the information system. They are divided into two main groups :

Structural diagrams or static diagrams (UML Structure)

- Class diagram
- object diagram (Object diagram)
- Component diagram
- Deployment diagram
- Package diagram
- Composite structure diagram

Behavioural diagrams or dynamic diagrams (UML Behavior)

- Use case diagram (Use case diagram)
- Activity diagram (Diagramme d'activités)
- State machine diagram (State machine diagram)
- **Interaction diagrams (Interaction diagram)**
    - Sequence diagram
    - Communication diagram
    - Interaction overview diagram
    - Timing diagram

For a model, these diagrams are not necessarily all produced. In this course we will study 9 of the 13 diagrams: use case, class, object, transition state, activity, collaboration, sequence, component, and deployment diagrams.

1. Use Case Diagram

This is the first diagram of the UML model, the one where the relationship between the user and the objects that the system implements is ensured. It is used to collect, analyze and organize requirements, and to identify the major functionalities of a system. It is therefore the first UML step in the analysis of a system. It captures the behavior of a system as an external user sees it. It divides the functionality of the system into coherent units: use cases, which make sense to the actors. Use Cases are used to express the needs of the users of a system. To develop use cases, interviews with users are required.

1.1 the elements of a use case diagram

There are 2 main concepts in the construction of a use case diagram: the actors and use cases.

1.1.1 Actor

An actor is the idealization of a role played by an external person, process or thing that interacts with a system. It is represented by a little man with his name (his role) written underneath. It is also possible to represent an actor in the form of a folder.

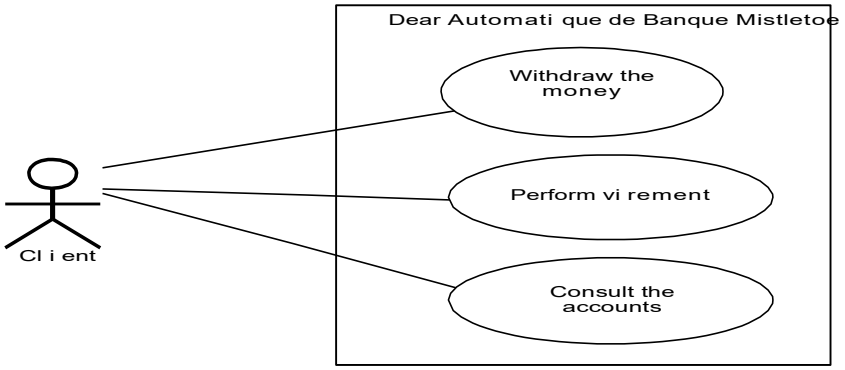
1.1.2 Use cases

A use case is a coherent unit representing a functionality visible from the outside. It performs an end-to-end service, with a trigger, a sequence and an end, for the actor who initiates it. It therefore models a service provided by the system, without imposing the way in which this service is provided.

service. It is represented by an ellipse containing the name of the case (infinitive verb), and optionally, above the name, a stereotype.

1.1.3 Representation of a use case diagram

The system boundary is represented by a frame. The name of the system is inside the frame at the top. The actors are on the outside and the use cases on the inside.



1.2 Relationships in use case diagrams

A distinction is made between relationships between actors and use cases, between actors and between use cases.

1.2.1 Relationship between actors and use cases

Between an actor and a use case there is a relationship of association, which is a communication path between the actor and this use case. It is represented by a continuous line.

Main and secondary actors

An actor can be a primary or secondary actor. It is qualified as primary for a use case when this use case is of service to this actor. The other actors are then qualified as secondary. A use case has at most one main actor. A primary actor obtains an observable result of the system while a secondary actor is solicited for additional information. In general, the main actor initiates the use case by his solicitations. The stereotype "primary" or "secondary" is used.

Internal use cases

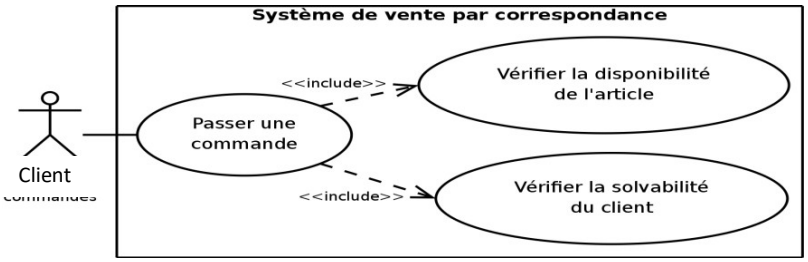
When a case is not directly related to an actor, it is qualified as an internal use case.

1.2.2 Relationship between use cases

A relationship between two use cases can be defined on a case-by-case basis.
A distinction is made between inclusion, extension and generalization/specialization relationships.

Inclusive relationship

A case A includes a case B if the behavior described by case A includes the behavior of case B: case A depends on B. When A is solicited, B is obligatorily solicited, as a part of A. This dependence is symbolized by the stereotype "*include*". For example, access to bank account information necessarily includes an authentication phase with an identifier and a password.
Inclusions essentially allow factoring a part of the description of a use case that would be common to other use cases.

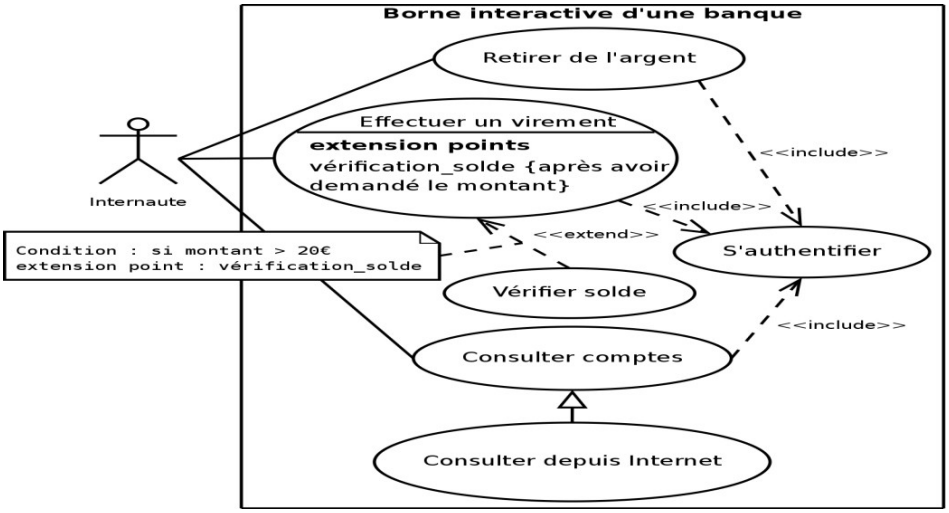


Extension relationship

A use case A is said to extend a use case B when use case A can be called during the execution of use case B. Running B may possibly result in the execution of A: unlike inclusion, extension is optional. This dependency is symbolized by the stereotype "*extend"*. An extension is often conditional. Graphically, the condition is expressed in the form of a note.

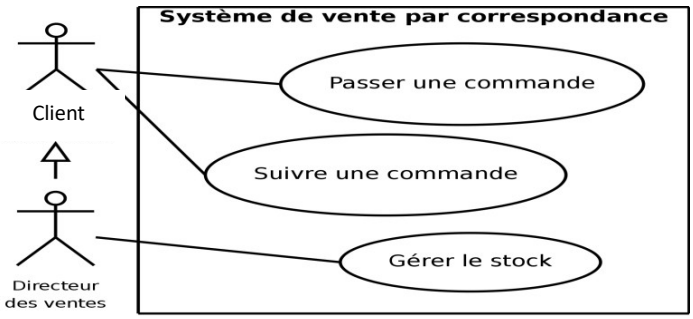
Generalization relationship

A case A is a generalization of a case B if B is a special case of A. In the figure below, the Consultation of an account via the Internet is a special case of consultation.
For example, the figure below shows that the sales manager is an Internet user with an additional power: in addition to being able to carry out operations online, he can manage stocks.



1.2.3 Relationships between actors

The only possible relationship between two actors is generalization: an actor A is a generalization of an actor B if the actor A can be substituted by the actor B. In this case, all use cases accessible to A are also accessible to B, but the reverse is not true.



2.3 Textual description of use cases

The use case diagram describes the major functions of a system from the perspective of actors, but does not detail the dialogue between actors and use cases. It is

therefore recommended to write a textual description of the use case, which usually consists of three parts.

The first part allows to identify the case :

Name: Use an infinitive turn (e.g. Receiving a package).

Purpose: A summary description to understand the main intent of the use case. This part is often filled in at the beginning of the project in the use case discovery phase.

Main actors: Those who are going to realize the use case (the relationship with the use case is illustrated by the line linking the use case and the actor in a use case diagram)

Secondary actors: Those who only receive information after the case has been completed of use

Dates : The dates of creation and update of the current description.

Person in charge: The name of the person in charge.

Version: The version number.

The second part contains a description of how the case works in the form of an sequence of messages exchanged between the actors and the system.

There is always a nominal sequence describing the normal course of the case, to which are frequently added alternative sequences (branches in the nominal sequence) and exception sequences.

The preconditions: they describe what state the system (application) must be in before this case of use can be triggered.

Scenarios: described in the form of exchanges of events between the actor and the system. On distinguishes the nominal scenario, which takes place when there is no error, from the alternative scenarios which are the variants of the nominal scenario and finally the exception scenarios which describe the cases of errors.

Postconditions: They describe the state of the system at the end of the different scenarios.

The third part of the description of a use case is an optional topic. It generally contains non-functional specifications (technical specifications, . . .).

2. Class Diagram

The class diagram is considered to be the most important part of object-oriented modeling, it is the only one required for such modeling. While the use case diagram shows a system from the point of view of the actors, the class diagram shows its internal structure. It provides an abstract representation of the system objects that will interact together to realize use cases. The same object can very well be involved in the realization of several use cases. It is a static view because the time factor is not taken into account in the system behavior. The class diagram models the concepts of the application domain as well as the internal concepts created from scratch during the implementation of an application.

The main elements of this static view are the classes and their relationships: association, generalization and several types of dependencies, such as realization and use.

3.1 Classes

3.1.1 Class instance

An instance is a concretization of an abstract concept. For example:

- the ML 4MATIC is an instance of the abstract concept ***car*** ;
- The friendship between Paul and Mireille is an instance of the abstract concept ***Amitié*** ;

A class is an abstract concept representing various elements such as :

- concrete elements (e.g. student, planes, car,...),
- abstract elements (e.g. orders for goods or services),
- Components of an application (e.g., dialog box buttons), etc. Any object-oriented system is organized around classes.

A class is the formal description of a set of objects having semantics and common characteristics.

An object is an instance of a class. It is a discrete entity with an identity, state and behavior that can be invoked. For example, if we consider that ***Man*** is a class, we can could say that the person ***francky*** is an instance of Man, that is to say an object.

3.1.2 Characteristics of a class

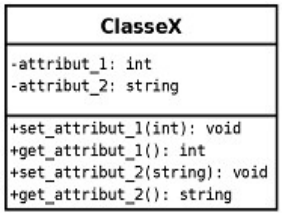
A définit class of objects with common characteristics. The characteristics of an object allow spécifier to determine its state and behavior.

- **State of an object**: These are the attributes and generally the association endings, all two gathered under the term of structural properties, or simply properties, which describe the state of an object. Associations are used to connect classes in the class diagram; in this case, the termination of the association (on the target class side) is usually a property of the base class. Properties described by attributes take on values when the class is instantiated. The instance of an association is called a link.
- **Object behavior**: Operations describe the individual elements of an **object.** behaviour that can be invoked. These are functions that can take input values and modifier attributes or produce results. An operation is the spécification (declaration) of a method.

Attributes, association endings and methods are therefore the characteristics of a class (and its instances).

3.1.3 Graphic representation

A class is represented by a rectangle generally divided into three compartments. The first indicates the name of the class, the second its attributes and the third its operations.



3.1.4 Encapsulation, visibility, interface

Encapsulation allows définir levels of visibility of the elements of a container. The visibility declares the possibility for a modeling element to reference an element that is located in a different namespace than the element that establishes the reference. It is part of the relationship between an element and the container that hosts it, which may be a package, a class or another namespace. There are four visibilities prédéfinies.

- ***Public*** or +: any element that can see the container can also see the indicated element.
- ***Protected*** or #: only an element located in the container or one of its descendants can see the indicated item.
- ***Private*** or - : only an element located in the container can see the element.
- ***Package*** or ~ or nothing : only an element declared in the same package can see the element. In a class, the visibility marker is located at the level of each of its characteristics (attributes, association endings and operation). It allows to indicate if another class can y access. In a package, the visibility marker is located on elements contained directly in the package, such as classes, nested packages, etc. It indicates whether another package that can access the first package can see the items.

3.1.5 Name of a class

The name of the class should evoke the concept described by the class. It begins with a capital letter. Other information can be added such as the name of the author of the modeling, the date, etc.
The basic syntax of the declaration of a class name is as follows:
[< Package_Name_1> ::... ::< Package_Name_1>] < Class_Name> [{ [abstract], [<author>], [<date>], ... }]

3.1.6 Attributes

Class Attributes

Attributes define information that a class or object must know. They represent the data encapsulated in the objects of this class. Each piece of information is defined by a name, a data type, a visibility and can be initialized.

Class Attributes

By default, each instance of a class has its own copy of the class attributes.
Attribute values may therefore differ from one object to another. However, it is sometimes necessary to define a class attribute (static in Java or C++) that keeps a unique value shared by all instances of the class. Instances have access to this attribute but do not have a copy of it. A class attribute is therefore not a property of an instance but a property of the class and access to this attribute does not require the existence of an instance. Graphically, a class attribute is underlined.

Derived Attributes

Derived attributes can be calculated from other attributes and formulas. The derived attributes are symbolized by adding a "/" in front of their name.

3.1.7 Methods

Method of the class

In a class, an operation (same name and same type of parameters) must be unique.
When the name of an operation appears several times with different parameters, the operation is said to be overloaded. On the other hand, it is impossible for two operations to be distinguished only by their returned value.
The declaration of an operation contains the types of the parameters and the type of the return value, its syntax is as follows:
< visibility> <method_name> ([<parameter_1>, ... , <parameter_N>]) : [< returned_type>] [{< properties> }]
The syntax for defining a parameter (<parameter>) is as follows:
[<direction>] <parameter_name> :< type> [' [' [' < multiplicity> ']]] =< default_value>]
The direction can take one of the following values :
in: Input parameter passed by value. Parameter changes are not available to the caller. This is the default behavior.
out: Output parameter only. There is no input value and the final value is available.
for the caller.
inout: Input/output parameter. The final value is available to the caller.
The type of the parameter (< type>) can be a class name, an interface name or a predefined data type. The properties (<properties>) correspond to constraints or additional information such as exceptions, preconditions, postconditions or the indication that a method is abstract (abstract keyword), etc.

Class method

As for class attributes, it is possible to declare class methods. A class method can only manipulate class attributes and its own parameters. This method does not have access to the attributes of the class (of the instances of the class). Access to a class method does not require the existence of an instance of the class.
Graphically, a class method is highlighted.

Methods and abstract classes

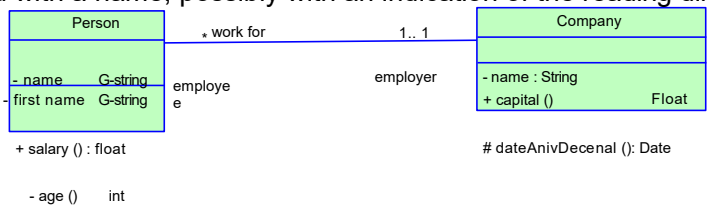
A method is said to be abstract when we know its header but not the way it can be realized (we know its statement but not its definition).
A class is said to be abstract when it defines at least one abstract method or when a parent class contains an abstract method that has not yet been realized.
We cannot instantiate an abstract class: it is destined to specialize. An abstract class may very well contain concrete methods.

A pure abstract class consists only of abstract methods. In object-oriented programming, such a class is called an interface. We use the steriotype <abstract>.

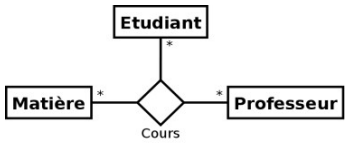
3.2 Relationships between classes

3.2.1 Association

An association is a relationship between two or more classes (binary association) or more classes (association
n-aire), which describes the structural connections between their instances. It indicates that there can be links between instances of associated classes.
A binary association is materialized by a solid line between the associated classes. It can be ornamented with a name, possibly with an indication of the reading direction.



When both ends of the association point to the same class, the association is said to be reflexive. An n-ary association links more than two classes. The dotted line of a class-association can be connected to the diamond by a broken line to represent an n-ary association with attributes, operations or associations. An n-ary association is represented by a large lozenge with a path leading to each participating class. If the association has a name, it is placed next to the diamond.

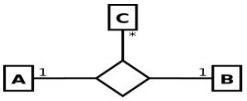


3.2.2 Multiplicity or cardinality

The multiplicity associated with a termination of association, aggregation or composition declares the number of objects that can occupy the position defined by the association termination.
Here are some examples of multiplicity:

- exactly one : 1 or 1 1
- several : * or 0. *
- at least one: 1. *
- from one to six: 1. 6

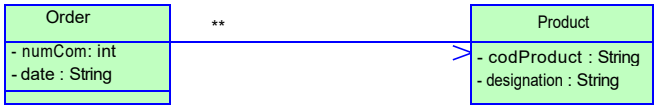
In a binary association, the multiplicity on the target termination defines the number of objects of the target class that can be associated to only one given object of the source class. In an n-area association, the multiplicity appearing on the link of each class applies to one instance of each class, excluding the class-association and the considered class. For example, if we take a ternary association between classes (A, B, C), the multiplicity of the ending C indicates the number of objects C that can appear in the association with a particular pair of objects A and B.



Note 1: For an n-aire association, the minimum multiplicity must in principle, but not necessarily, be 0. Indeed, a minimum multiplicity of 1 (or more) on one end implies that there must be a link (or more) for ALL possible combinations of the instances of the classes located at the other ends of the n-aire association!
Note 2: For those familiar with the entity/relationship model, multiplicities are "upside down" in UML. (by reference to Merise) for binary associations and "right-sided" for n-ary with n > 2.

3.2.3 Airworthiness

Airworthiness indicates whether it is possible to cross an association. Navigability is represented graphically by an arrow on the side of the navigable termination. By default, an association is navigable in both directions.



In this example the instances of the class Product do not store a list of objects of type Order. Conversely, each order object contains a list of products (navigable ending on the side of the Product class).

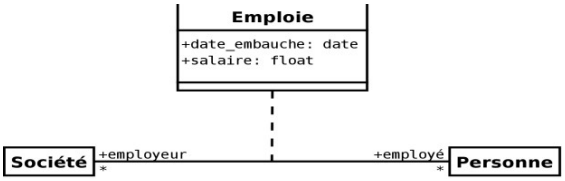
3.2.4 Qualification

When a class is linked to another class by an association, it is sometimes preferable to restrict the scope of the association to a few targeted elements (such as one or more attributes) of the class. These targeted elements are called a qualifier. The object selected by the value of the qualifier is called the target object. The association is called qualified association. A qualifier always acts on an association whose multiplicity is several on the target side.

A qualified object and a qualifier value generate a unique linked target object. When considering a qualified object, each qualifier value designates a unique target object.
For example, the diagram above tells us that :
- An account in a bank belongs to no more than two people. In other words, one instance of the couple {Bank, account} is associated with zero to two instances of the class Person.
- But a person can have several accounts in several banks. That is to say that a instance of the class Person can be associated to several (including zero) instances of the pair. {Bank, account}.
- Of course, and in any case, an instance of the couple {Person, account} is in association with a single instance of the Bank class.

3.2.5 Class-association

A class-association has the characteristics of associations and classes, and is used when an association must have properties. For example, the association ***Employment*** between a company and an individual has as properties the salary and the date of hire. It is characterized by a dashed line between the class and the association it represents.



NB: It is not possible to attach a class-association to more than one association.

Aggregation and composition

Aggregation

Association representing a structural or behavioral inclusive relationship of an element in a set.
In an association, neither class is more important than the other. If one wishes to model an *all/part* relationship where a class is a larger (all) compound element of smaller elements (parts), an aggregation must be used.
Graphically, an empty diamond is added on the side of the aggregate. Contrary to an association simple, aggregation is transitive. The meaning of this simple form of aggregation is only conceptual. It does not lead to a constraint on the lifetime of the parts in relation to the whole.

Composition

The composition, also called composite aggregation, describes a structural capacity between instances. Thus, the destruction of the composite object implies the destruction of its components. A

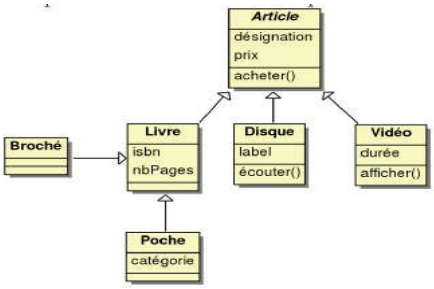
instance of the part always belongs to at most one instance of the composite element: the multiplicity of the composite side must not be greater than 1 (i.e. 1 or 0.. 1).
Graphically, a solid diamond is added on the side of the aggregate.

Generalization and Legacy

Inheritance allows the classification of objects. Generalization describes a relationship between a general class (base class or parent class) and a specialized class (sub-class). The specialized class has all the characteristics of the base class, but contains additional information (attributes, operations, associations).
The main properties of inheritance are :

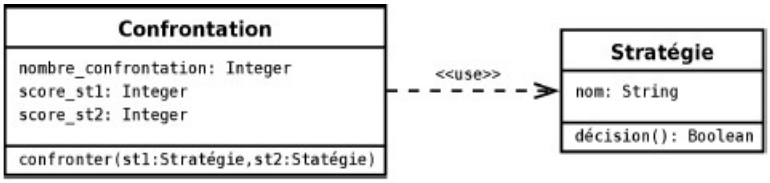
- The child class has all the characteristics of its parent classes.
- A child class can redefine (same signature) one or more methods of the parent class.
- All associations of the parent class apply to the derived classes.
- An instance of a class can be used anywhere an instance of its parent class is expected.
- A class can have several parents, so we speak of multiple inheritance.

NB: The C++ language allows its implementation, but not the Java language.



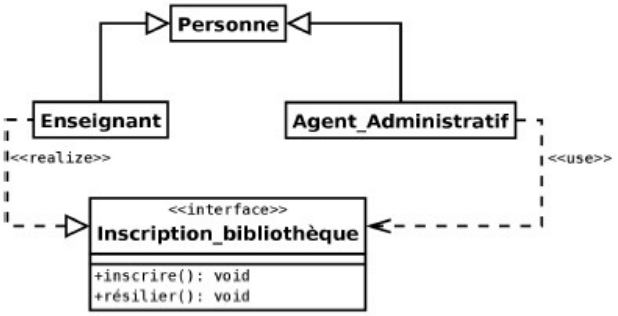
Dependency

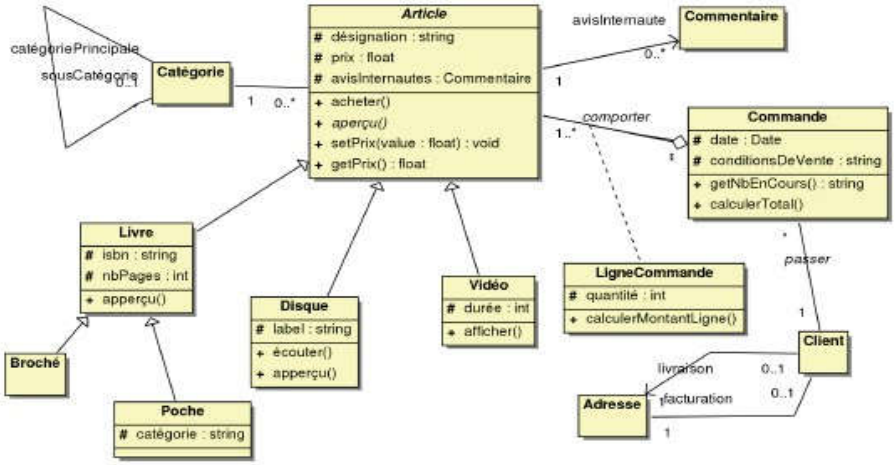
A dependency is a unidirectional relationship expressing a semantic dependency between elements of the model. It is represented by an oriented discontinuous line. It indicates that the modification of the target may imply a modification of the source. The dependency is often stereotyped. A dependency is often used when one class uses another as an argument in the signature of an operation.
For example, the diagram below shows that the Confrontation class uses the Strategy class because the Confrontation class has a ***confrontation*** method of which two parameters are of the type Strategy.



Interfaces

It is a workbook, stereotypical "*interface*", whose role is to group together a set of properties and operations ensuring a consistent service. An interface is represented as a class except for the addition of the "interface" stereotype.
An interface must be realized by at least one class and can be realized by several classes. Graphically, this is represented by a broken line ending with a triangular arrow and the stereotype "*realize*". A class can very well realize several interfaces. A class (client class of the interface) can depend on one interface (interface required). This is represented by a dependency relationship and the stereotype "*use*". Beware of conflict problems if a class depends on an interface realized by several other classes.





3. Object diagram

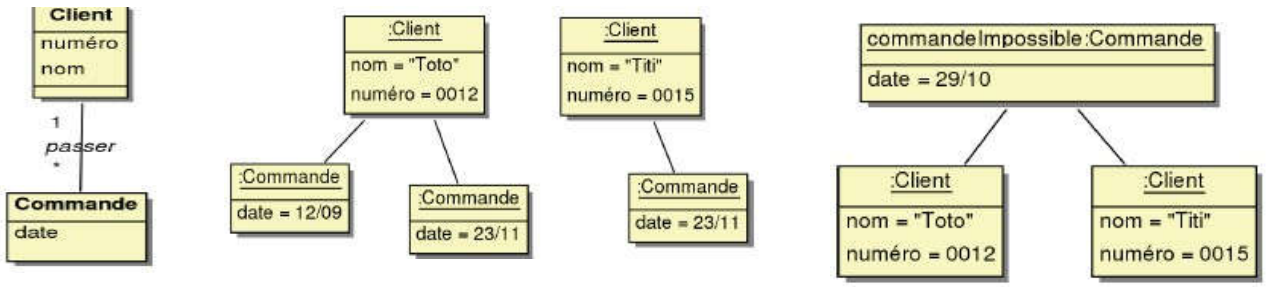
4.1 Presentation

An object diagram represents objects (class instances) and their links (relationship instances) to give a fixed view of the state of a system at a given time. It can be used for :
- illustrate the class model by showing an example that explains the model ;
- clarify certain aspects of the system by highlighting imperceptible details in the class diagram ;
- express an exception by modeling special cases or non generalizable knowledge that are not modeled in a class diagram ;
- take a snapshot of a system at a given time.

The class diagram models rules and the object diagram models facts.
An object diagram does not show the evolution of the system over time. To represent a interaction, a communication or sequence diagram should be used.

4.2 Representation

The figure below shows : (A) a class diagram; (B) an object diagram consistent with this class diagram; (C) an object diagram consistent with this class diagram.
class diagram; (C) an object diagram inconsistent with the class diagram.

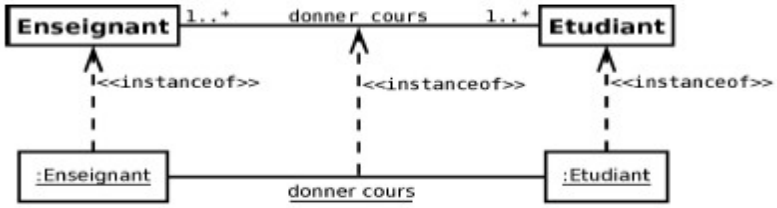


A B C

Graphically, an object is represented as a class. However, the operations compartment is not useful. Moreover, the name of the class of which the object is an instance is preceded by a ":" and is underlined. To differentiate objects of the same class, their identifier can be added before the name of the class. Finally the attributes receive values. When some attribute values of an object are not filled in, it is said that the object is partially defined.
The generalization relation does not have an instance, so it is never represented in a object diagram. Graphically, a link is represented as a relationship, but if there is a name, it is underlined. Naturally, multiplicities are not represented.

4.3 Instantiation dependency relationship

It is stereotyped "instanceof" and describes the relationship between a workbook and its instances. In particular, it links links to associations and objects to classes.



4. State-transition diagram

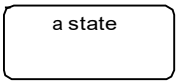
5.1 Presentation

UML state-transition diagrams describe the internal behavior of an object using a finite state machine. They present the possible sequences of states and actions that a class instance can process during its life cycle in response to events. They usually specify the behavior of a class instance. But sometimes also the internal behavior of other elements such as use cases, subsystems, methods.
This diagram gathers and organizes the states and transitions of a given class. It is advisable to build a state-transition diagram for each class that has an important dynamics. It can only be associated to one class.

5.2 Elements of a state-transition diagram

5.2.1 Status

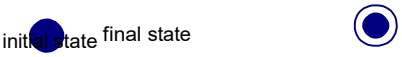
A state represents a period of time in the life of an object during which it is waiting for an event or performing an activity. An object can go through a series of states during its lifetime. It is represented by a rectangle with rounded corners.



Some states, called composite states, can contain (wrap) sub-states.
The name of the state can be specified in the rectangle and must be unique in the state diagram. transitions, or in the enveloping state. A state can be anonymous.

5.2.2 Initial and final status

These are pseudo states. The first one indicates the starting state, by default, when the state-transition diagram is invoked. When an object is created, it enters the initial state.
The second indicates that the state-transition diagram, or wrap-around state, is complete.



5.2.3 Event

An event is something that occurs during the execution of a system and must be modeled. Transitions in a transition state diagram are triggered by triggering events. They occur at a specific time and have no duration. When an event is received, a transition can be triggered and switch the object to a new state. There are several types: signal, call, change and temporal.

a. Signal type event (signal): A signal is a type of event intended to convey one-way asynchronous communication between two objects. The sender object explicitly creates and initializes a signal instance and sends it to an object or group of objects. The reception of a signal is an event for the recipient object. The syntax of a signal is as follows:

event_name ([parametrize : type])

b. Call event (call): A call event represents the receipt of the call of an operation by an object. These are operations declared at the clad diagram level. The parameters of the operation are those of the call event. The syntax of a call event is the same as that of a signal. On the other hand, call events are methods declared at the class diagram level. **event_name ([parametrize : type])**

c. Change event (change): it is generated by the satisfaction (change from false to true) of a Boolean expression on attribute values. It is a declarative way of waiting for a condition to be satisfied. The syntax of a change event is as follows: **when (condition_bolean)**

d. Temporal event (after): they are generated by the passage of time. They are specified either absolutely (precise date) or relatively (elapsed time). By default, the time starts to elapse as soon as the current state is entered.

The syntax of a relatively specified time event is as follows:
after (duration)
An absolutely specified time event is defined using a change event: **when (date = < date>)**

5.2.4 Transition

A transition defines the response of an object to the occurrence of an event. It generally links two states, E1 and E2, and indicates that an object in an E1 state can enter the E2 state and perform certain activities, if a trigger event occurs and the guard condition is verified.
The syntax of a transition is as follows: **nameEvent (params) [guard] / activite**
The same event can be the trigger for several transitions leaving the same state. Each transition with the same event must have a different guard condition.

a. Custody condition

Custody is a condition that must be met in order to trigger the transition. The activity designates instructions to be carried out at the time of shooting.

It is a logical expression on the attributes of the object, associated with the state-transition diagram, as well as the parameters of the triggering event. It is evaluated only when the triggering event occurs. If the expression is false, the transition does not trigger, otherwise the transition is triggered and its effects occur.

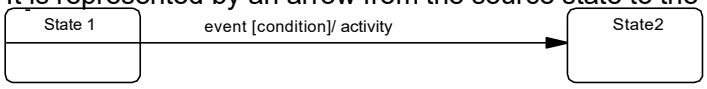
b. Effect of a transition

When a transition is triggered, its effect (specified by ' /' < activity> in the syntax) is executed. It is usually an activity that can be
- a primitive operation such as an assignment instruction ;
- sending a signal ;
- the call of an operation ;
- a list of activities, etc.

The way of specifying the activity to be performed is left free (natural language or pseudo-code). When the execution of the effect is finished, the target state of the transition becomes active.

c. External Transition

An external transition is a transition that changes the active state. This is the most common type of transition.
widespread. It is represented by an arrow from the source state to the target state.



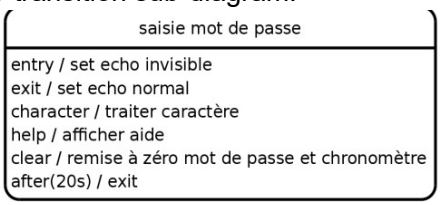
d. Completion Transition

A transition with no explicit trigger event is triggered at the end of the activity contained in the source state (including nested states). It can contain a guard condition that is evaluated at the time the activity contained in the state ends. These transitions are, for example, used to connect initial states with their successor states.

e. Internal Transition

The rules for triggering an internal transition are the same as for an external transition except that an internal transition does not have a target state and the active state remains the same following its triggering. On the other hand, internal transitions are not represented by arcs but are specified in a compartment of their associated state.

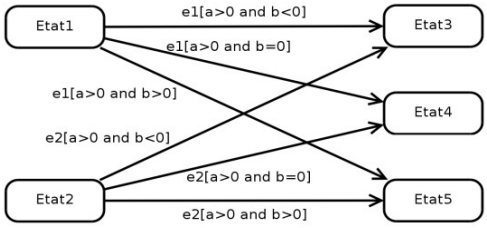
Internal transitions have predefined event names corresponding to particular triggers: *entry, exit, do* and *include*. These reserved keywords take the place of the event name in the syntax of an internal transition.
entry: Allows you to specify an activity that is performed when you enter the state.
exit: Allows you to specify an activity that is performed when exiting the state.
do: A do activity starts as soon as the entry activity is completed. When this activity is finished, a completion transition can be triggered, after the exit activity has been executed of course. **include:** Allows to invoke a state-transition sub-diagram.

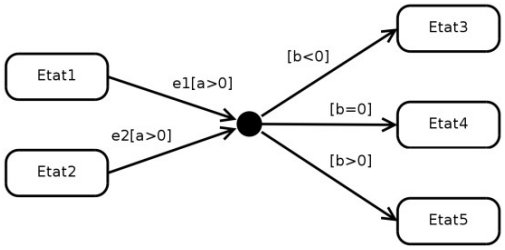


5.2.5 Point of choice

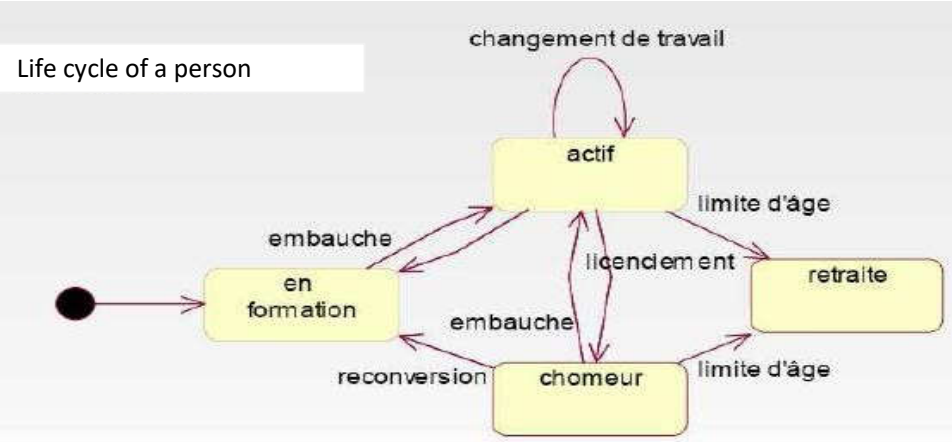
To represent alternatives for crossing a transition, particular pseudo states are used: the junction points (represented by a small solid circle) and the decision points (represented by a diamond).

A junction point can have several incoming and several outgoing transition segments. However, it cannot have internal activity or outgoing transitions with event triggers.





A decision point has one input and at least two outputs. Guards located after the decision point are assessed when the decision point is reached. This makes it possible to base the choice on the results obtained by crossing the segment before the decision point. It is possible to use a particular guard, noted [else], on one of the segments downstream of a choice point. This segment can only be crossed if the guards of the other segments are all false.



5. Activity Diagram

6.1 Presentation

Activity diagrams are used to graphically represent the behavior of a method or the progress of a use case. They focus on treatments. Unlike state-transition diagrams, activity diagrams are not specifically linked to a particular workbook. In the design phase, activity diagrams are particularly well suited to describing use cases.

5.2 Concepts

6.2.1 Action

The notion of action is to be compared to the notion of elementary instruction of a programming language (C++, Java). An action is the smallest treatment that can be expressed in UML. An action can be, for example :

- an assignment of value to attributes ;
- the creation of a new object or link ;
- a simple arithmetic calculation ;
- the emission of a signal ;
- the reception of a signal ;

5.2.2 Activity

An activity defines a behavior described by an organized sequence of units whose simple elements are actions. The execution flow is modeled by nodes connected by arcs (transitions).

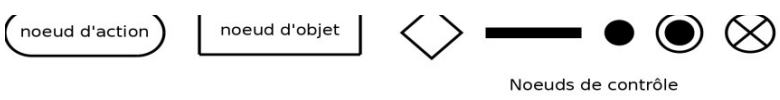
5.2.3 Activity node

An activity node is an element to represent the steps of an activity. There are three families of activity nodes :

Executable or executable: activity node that can be executed.

- Object nodes: Allows you to define an object flow (i.e. a data flow) in an activity diagram. This node represents the existence of an object generated by an action in an activity and used by other actions.

control nodes :

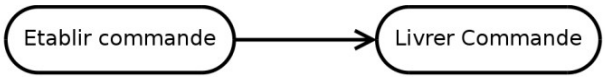


From the left to the right, we find :

The node representing an action, which is a variety of execution node, object node, decision or merge node, bifurcation or union node, start node, end node and flow end node.

5.2.4 Transition

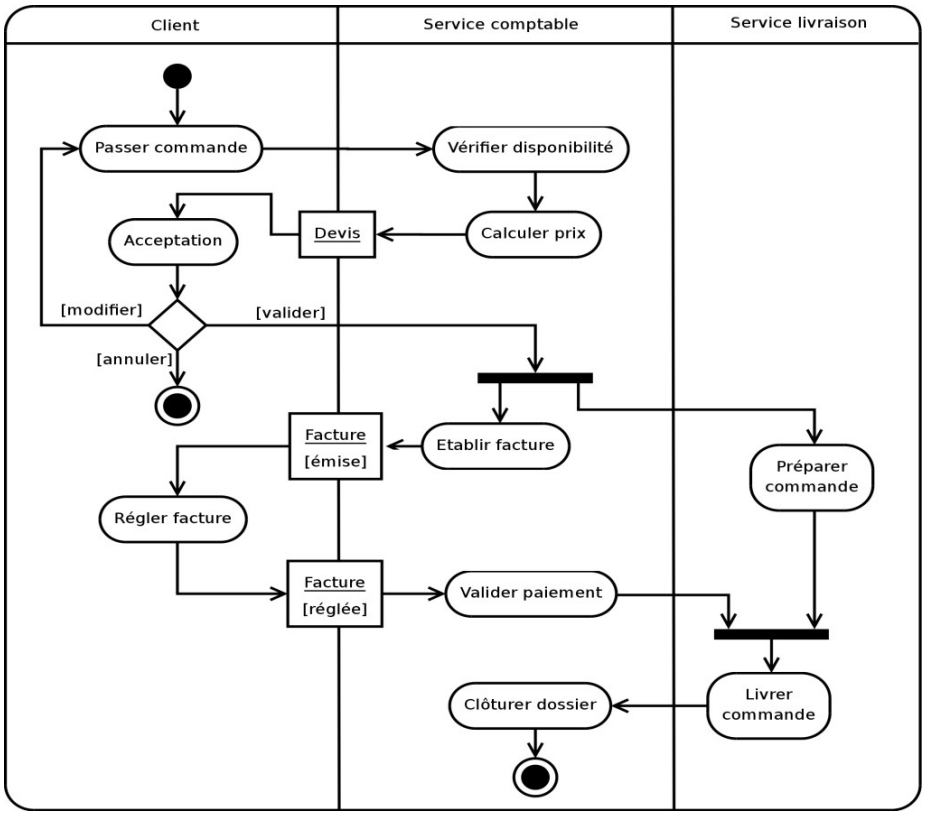
It materializes the passage from one node of activity to another. Graphically, transitions are represented by solid arrows that connect activities (nodes) together. They are triggered as soon as the source activity is completed and automatically and immediately trigger the start of the next activity to be triggered (the target activity).



5.2.5 Scores

Partitions, often referred to as activity corridors or waterlines because of their notation, are used to organize activity nodes in an activity diagram by grouping them together. They can, for example, be used to specify the class responsible for implementing a set of tasks. In this case, the class in question is responsible for implementing the behavior of the nodes included in the said partition.
They are represented by continuous lines. These are usually vertical lines, but they can be horizontal or even curved. Transitions can, of course, cross the boundaries of the scores.

5.3 Representation



6. Interaction diagrams

An object interacts to implement a behavior. This interaction can be described in two complementary ways: one is centered on individual objects (state-transition diagram) and the other on a collection of objects that cooperate (interaction diagrams). The interaction diagram provides a more global view of the behavior of a set of objects.

A distinction is made between: the *communication diagram,* which focuses on the structural organization of the objects that send and receive messages, and the *sequence diagram,* which focuses on the chronology of the sending of messages. They provide a link between use case diagrams and class diagrams: they show how objects communicate to achieve a certain functionality. To produce an interaction diagram, one must focus on a subset of system elements and study how they interact to describe a particular behavior.

6.1 Collaboration or Communication Diagram

A collaboration allows to describe the implementation of a functionality by a set of participants. A role is the description of a participant. For example, to implement a use case, a set of classes, and other elements, working together to achieve the behavior of that use case, must be used. This set of elements, comprising both a static and a dynamic structure, is modeled in UML through collaboration. This diagram is often used to illustrate a use case or to describe an operation. The communication diagram helps to validate the associations of the class diagram by using them as a message transmission medium.

7.1.1 Participants

Participants are represented by rectangles containing a label with the syntax : **[< role_name>] : [< Type_Name>].** At least one of the two names must be specified in the label, the colon (:) is mandatory.

7.1.2 Connectors

Relationships between representatives are called connectors and are defined by a solid line.

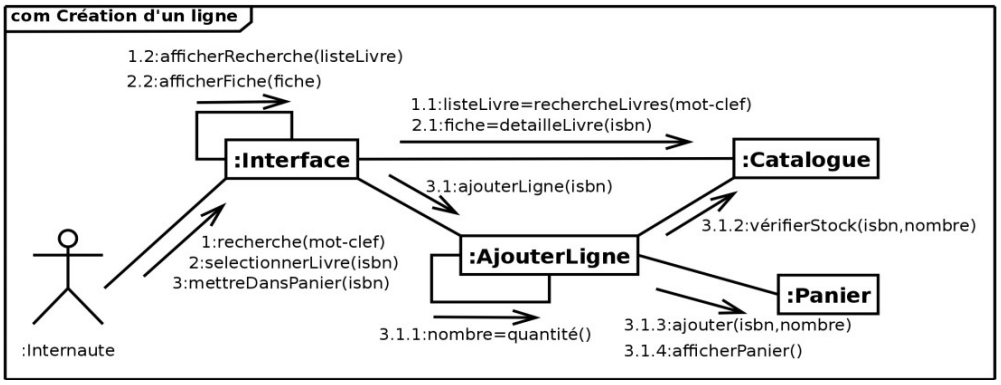
7.1.3 Messages

In a communication diagram, messages are usually ordered in an ascending sequence number. A message is usually specified as follows:

[' ['< cond> ']' [< séq>] [*[||] [' ['< iter> ']']] :] [< var> :=] < msg> ([< by>])

< ***cond>*** is a condition in the form of a boolean expression in square brackets.
< seq> is the sequence number of the message. Messages are numbered by sending and sub-sending, designated by numbers separated by dots: thus the sending of message 1.4.4 is later than the sending of message 1.4.4.
of message 1.4.3, both of which are consequences (sub-sends) of receiving a message
1.4. Simultaneous sending is designated by a letter: messages 1.6a and 1.6b are sent at the same time.
< iter> specifies (in natural language, in square brackets) the sequential (or parallel, with ||) sending of
several messages. You can omit this specification and keep only the character * to designate a recurring message sent a certain number of times.
< var> is the return value of the message, which will for example be transmitted as a parameter to another one.
message.
< msg> is the name of the
message.
< by> designates the (optional) parameters of the
message.

The diagram below shows the communication flowchart illustrating the search and then the addition of a book to your virtual shopping cart when ordering on the Internet.



6.2 Sequence diagram

The main information contained in a sequence diagram are the messages exchanged between the lifelines, presented in chronological order. Thus, contrary to the communication diagram, time is explicitly represented by a dimension
(The vertical dimension) and flows from top to bottom.

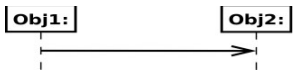
7.2.1 Representation of lifelines

A lifeline is represented by a rectangle, to which is hung a vertical dotted line, containing a label whose syntax is : **[< role_name>] : [< Type_Name>].**

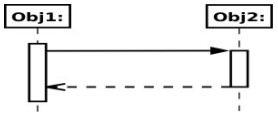
7.2.2 Message representation

A message defines a particular communication between lifelines. Several types of messages exist, the most common are :
- sending a signal ;
- the invocation of an operation;
- the creation or destruction of an instance.

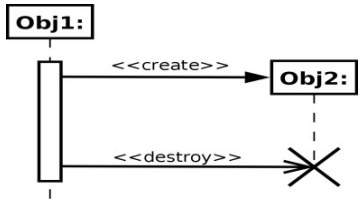
Asynchronous messages: They do not wait for a response and do not block the sender who does not know if the message will arrive at its destination, if so when it will arrive and if it will be processed by the recipient. A signal is, by definition, an asynchronous message.
Graphically, an asynchronous message is represented by an arrow with solid lines and at the end of the message, an asynchronous message is represented by an arrow with solid lines and at the end of the message, an asynchronous message.
from the lifeline of one object to the lifeline of another.



Synchronous messages: the sender remains blocked for the time it takes to process the message.
Graphically, a synchronous message is represented by an arrow with solid lines and a solid end from the lifeline of one object to the lifeline of another. This message can be followed by a response represented by a dotted arrow.

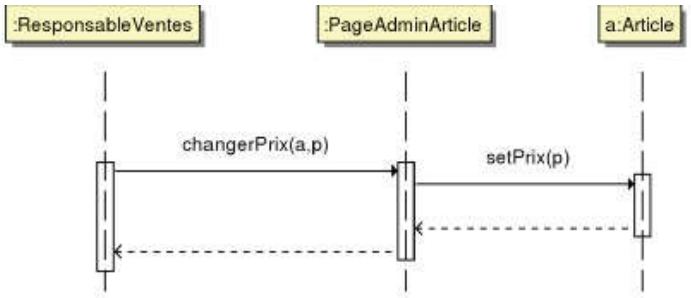


Instance creation and destruction messages: The creation of an object is materialized by an arrow pointing to the top of a lifeline. The destruction of an object is materialized by a cross that marks the end of the lifeline of the object. The destruction of an object is not necessarily consecutive to the reception of a message.

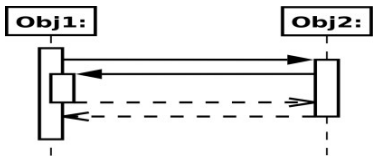


Events and messages: UML allows you to clearly separate the sending of the message, its reception, as well as the beginning of the execution of the reaction and its end.

Message and response syntax: In most cases, the reception of a message is followed by the execution of a method of a class. This method can receive arguments and the syntax of the messages allows these arguments to be transmitted.
The syntax for replying to a message is as follows:
[< attribute> =] message [: < return_value>]
Where message represents the sending message.



Method execution and active object: An active object initiates and controls the flow of activities. Graphically, the vertical dotted line of an active object is replaced by a double vertical line. A passive object, on the other hand, needs to be given the activity flow in order to execute a method. The specification of the execution of a reaction on a passive object is represented by a white or gray rectangle placed on the dotted lifeline. The rectangle can possibly bear a label.
Simultaneous executions on the same lifeline are represented by an overlapping rectangle.



7.2.3 Combined Interaction Fragments

A combined fragment represents joints of interactions, defined by an operator and operands. The operator conditions the meaning of the combined fragment. A fragment is represented in a rectangle whose upper left corner contains a pentagon. In the pentagon is the type of the combination, called interaction operator. The operands of an interaction operator are separated by a dotted line. The conditions of choice of the operands are given by boolean expressions in square brackets.
There are 12 interaction operators, including choice and loop operators: **alternative**, **option**, **loop,** and **break.**

Alt operator

The alternative operator, or ***alt***, is a conditional operator with several operands. It is somewhat equivalent to a multiple choice execution (switch condition in C++). Each operand has a guard condition. The absence of a guard condition implies a true condition. The ***other*** condition is true if no other condition is true. Exactly one operand whose condition is true is executed.

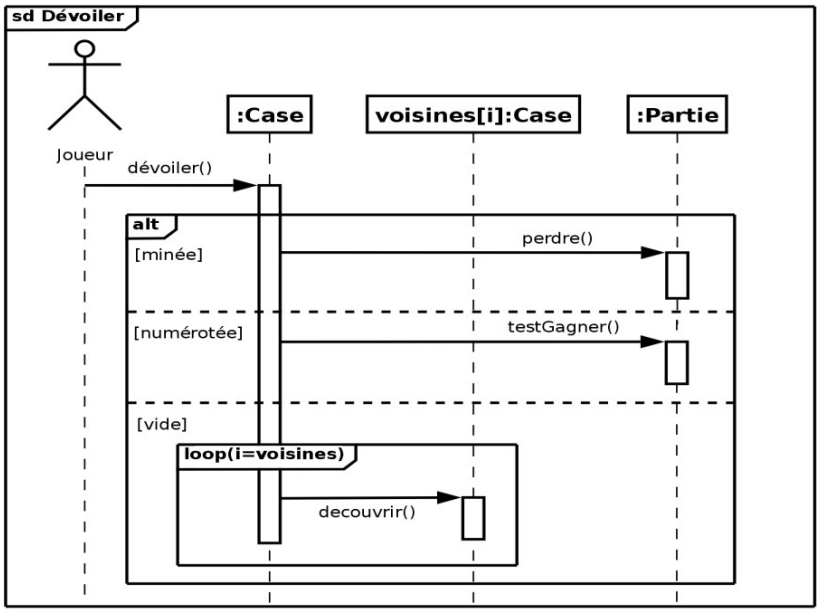
Opt operators

The option, or ***opt***, operator has an operand and an associated custody condition.
The subfragment executes if the custody condition is true and does not execute otherwise.

Loop operator

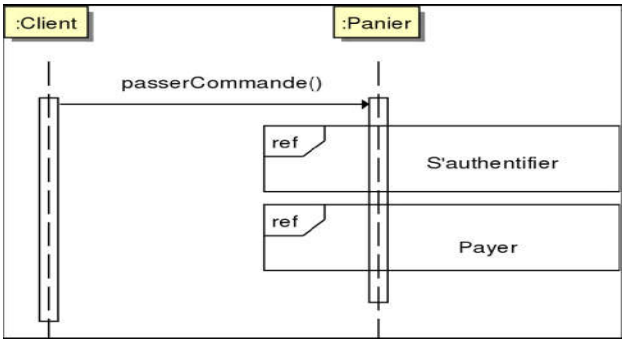
A combined ***loop*** fragment has a subfragment and specifies a minimum count and a maximum count. maximum (loop) and a guard condition.
The syntax of the loop is as follows: **loop[' (' < minInt> [',' < maxInt>] ')')**
The guard condition is placed between brackets on the lifeline. The loop is repeated at least ***minInt*** times before a possible Boolean guard condition is tested. As long as the condition is true, the loop continues, at most ***maxInt*** times.

Example



Operator ref

The operator ***ref***, allows to reuse an interaction. This consists in placing a fragment bearing the reference "ref" where the interaction is useful.



7. Component diagram

The notion of class, because of its low granularity and its fixed connections (associations with other classes materialize structural links), is not an appropriate response to the problem of reuse.

The component diagram represents the software architecture of the system. It allows structuring a software architecture at a lower level of granularity than classes. Components can contain classes. They also allow to specify the integration of third party software bricks. (EJB, CORBA, . Net, WSDL, etc... components).

8.1 Concept of component

A component is a stand-alone unit represented by a structured, stereotyped workbook. "Component", with one or more required or offered interfaces. Its internal behavior, usually realized by a set of classes, is totally hidden: only its interfaces are visible.

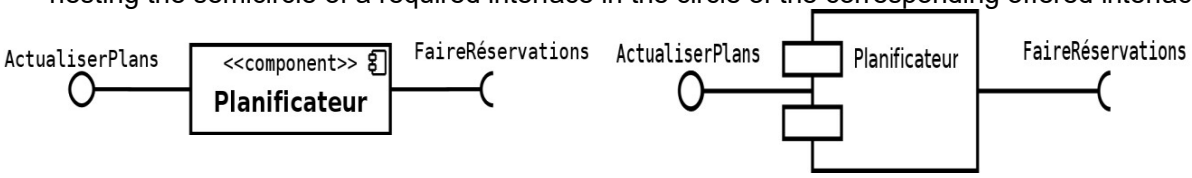
8.2 Notion of wearing

A port is a connection point between a folder and its environment. Graphically, a port is represented by a small square straddling the border of the workbook outline. The name of the port can be placed close to its representation.
Usually a port is associated with a required or offered interface. Sometimes it is directly connected to another port. The use of ports allows you to modify the internal structure of a workbook without affecting external clients.

8.3 Representation of a component diagram

The dependency relationship is used in component diagrams to indicate that an implementation element of one component uses the services offered by the implementation elements of another component.

When a component uses the interface of another component, the representation below can be used, by nesting the semicircle of a required interface in the circle of the corresponding offered interface.



8. Deployment Diagram

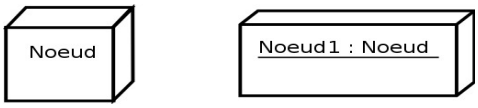
A system must run on hardware resources in a particular hardware environment. UML makes it possible to represent an execution environment as well as physical resources (with the parts of the system running on them) using deployment diagrams.

A node is a resource on which artifacts can be deployed for execution.

A deployment diagram describes the physical layout of the hardware resources that make up the system and shows the distribution of components on this hardware. Since each resource is materialized by a node, the deployment diagram specifies how the components are distributed on the nodes and what the connections between the components or nodes are.

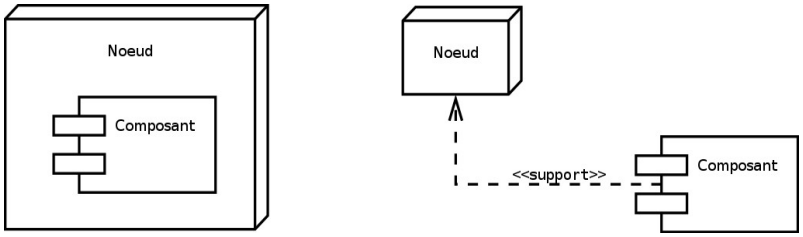
9.1 Representation of the nodes

Each resource is materialized by a node represented by a cube with a name. A node is a workbook and can have attributes (amount of memory, processor speed, etc.).



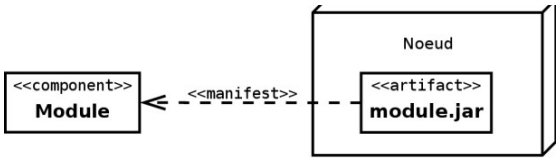
A particular node is an instance of node

Once the nodes have been defined, the components must be assigned to them. To show that a component is assigned to a node, one must either place the component in the node or link them by a stereotypical "support" dependency relationship oriented from the component to the node.



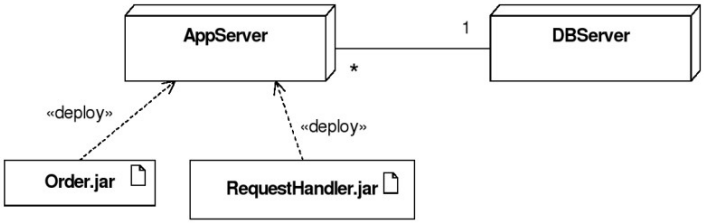
9.2 Notion of artifact (artifact)

An artifact corresponds to a concrete element existing in the real world (document, executable, file, database tables, script, etc.). It is represented as a folder by a rectangle containing the keyword "***artifact***" followed by the name of the artifact.
The implementation of models (classes, etc.) is done in the form of a set of artifacts. It is said that an artifact can manifest, i.e. implement, a set of model elements. The relationship between a model element and the artifact that implements it is called manifestation. Graphically, a manifestation is represented by a stereotypical dependency relationship "***manifest***".



An instance of an artifact is deployed on a node instance. Graphically, we use a stereotypical dependency relationship "***deploy***" pointing to the node in question. The artifact can also be included directly in the cube representing the node. Strictly speaking, only artifacts must be deployed on nodes. A component must therefore be manifested by an artifact which, itself, can be deployed on a node.

9.3 Example of a deployment diagram



Bibliography

- **UML 2, Laurent AUDIBERT,** Department of Computer Science, IUT de villetaneuse, Edition 2007-2008
- **Object Oriented Analysis and Design, B. LAVANYA,** Dept of computer Science, University of Madras, India.
- **Object modeling with UML, Pierre-Alain Muller , Nathalie Gaertner ,** Eyrolles.

Some Tools

Here are some tools used to build UML diagrams

- **Argo/UML** (http://argoumltigris.org/index.html) of the University of California UCI (www..uci.edu/pub/arch/uml)
- **Prosa/om** (www. prosa. fi/prosa.html) from Insoft Oy
- **Objecteering** (http://www.objecteering.com/) from Softeam (www.softeam.fr)
- **Paradigm Plus** (www.platinum.com/products/appdev/pplus_ps.htm) from Computer Associates
- **Rhapsody** (www. ilogix. com/fs_prod.htm) from I-Logix (www.ilogixcom)
- **Rose 2000** (http://www.rosearchitect.com/) from Rational Software Corporation (www.rational.com)
- **StP/UML** (www.aonix.com/Products/SMS/core7.1.html) from Aonix (www.aonix. fr)
- **Visual UML** (www.visualuml.com/products.htm) from Visual Object Modelers