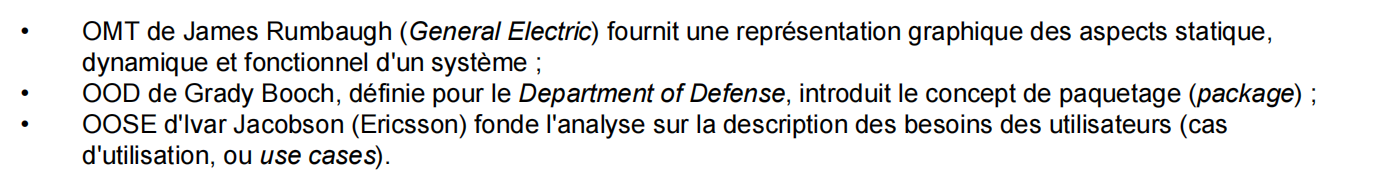
**Avant-propos :**

L'ordinateur électronique né des besoins de la guerre 🡪 Des langages plus évolués ont ensuite vu le jour comme Fortran en 1956 ou Cobol en 1959 🡪 c’est programmes se base sur les techniques de de branchement conditionnel et inconditionnel (goto) rendant les programmes extrêmement difficiles à développer , à maîtriser et à maintenir 🡪 La programmation structurée (Pascal en 1970, C en 1972, Modula et Ada en 1979…) a alors vu le jour et permis de développer et de maintenir des applications toujours plus ambitieuses + (l’algorithmique insuffisant 🡪 le génie logiciel est venu placer la méthodologie au cœur du développement logiciel 🡪 Des méthodes comme Merise (1978) se sont alors imposées).

La taille des applications ne cesse pas de croître 🡪 la programmation structurée a également rencontré ses limites, faisant alors place à la programmation orientée objet (Simula 67 en 1967, Smalltalk en 1976, C++ en 1982, Java en 1995…) 🡪 nécessité la conception de nouvelles méthodes de modélisation qui se base sur la notion d’objet🡪 la naissance de UML en 1990 (Unified Modeling Language) par la fusion de 3 méthodes OMT, Booch et OOSE qui était accepté par l'OMG (Object Management Group) en 1997



***Génie logiciel*** se charge d’appliquer des règles strictes encadrant la conception et permettant le travail en groupe et la maintenance du code afin de réussir le développement logiciel dans un contexte professionnel.

L'objectif du **génie logiciel** est de répondre á :

• l'augmentation des coûts ;

• les difficultés de maintenance et d'évolution ;

• la non-fiabilité ;

• le non-respect des spécifications

• le non-respect des délais.

53% du budget total d'un logiciel est affecté à la maintenance.

Ce coût est réparti comme suit :

• 34% maintenance évolutive (modification des spécifications initiales) ;

• 10% maintenance adaptative (nouvel environnement, nouveaux utilisateurs) ;

• 17% maintenance corrective (correction des bogues) ;

• 16% maintenance perfective (améliorer les performances sans changer les spécifications) ;

• 6% assistance aux utilisateurs ;

• 6% contrôle qualité ;

• 7% organisation/suivi ;

• 4% divers.

Le génie logiciel s'intéresse particulièrement à la manière dont le code source d'un logiciel est spécifié puis produit. Ainsi le génie logiciel touche au cycle de vie des logiciels :

• l'analyse du besoin,

• l'élaboration des spécifications,

• la conceptualisation,

• le développement,

• la phase de test,

• la maintenance.

Projet Logicielle dépasse 10 000 ligne 🡪 nécessite une équipe de développement bien structurée (organisée) 🡪 La gestion de projet se retrouve naturellement intimement liée au génie logiciel.

***La qualité du logiciel :***  ensemble des facteurs qui dépendent, entre autres

**Validité :**

Aptitude d'un produit logiciel à remplir exactement ses fonctions, définies par le cahier des charges et les spécifications.

**Fiabilité ou robustesse :**

Aptitude d'un produit logiciel à fonctionner dans des conditions anormales.

**Extensibilité (maintenance) :**

Facilité avec laquelle un logiciel se prête à sa maintenance, c'est-à-dire à une modification ou à une extension des fonctions qui lui sont demandées.

**Réutilisabilité :**

Aptitude d'un logiciel à être réutilisé, en tout ou en partie, dans de nouvelles applications.

**Compatibilité :**

Facilité avec laquelle un logiciel peut être combiné avec d'autres logiciels.

**Efficacité :**

Utilisation optimale des ressources matérielles.

**Portabilité :**

Facilité avec laquelle un logiciel peut être transféré sous différents environnements matériels et logiciels.

**Vérifiabilité :**

Facilité de préparation des procédures de test.

**Intégrité :**

Aptitude d'un logiciel à protéger son code et ses données contre des accès non autorisés.

**Facilité d'emploi :**

Facilité d'apprentissage, d'utilisation, de préparation des données, d'interprétation des erreurs et de rattrapage en cas d'erreur d'utilisation.

***C’est quoi un modèle :*** un langage commun, précis connu par les membres d’équipe qui permet de représenter un système d’information tout en réduisant sa complexité et en éliminant les détails qui n'affectent pas son comportement, alors c’est tout détail important pour la compréhension et la prédiction du système modélisé.

***Pourquoi modéliser :*** permet de mieux comprendre le fonctionnement du système et de maîtriser sa complexité et d'assurer sa cohérence afin de mieux répartir les tâches et également la réduction des coûts et des délais.

***Qui doit modéliser ?*** IL est préférable que la modélisation soit réalisée par la maîtrise d'ouvrage (MOA) de sorte que le métier soit maître de ses propres concepts. La MOE doit intervenir dans le modèle lorsque, après avoir défini les concepts du métier, pour introduire les contraintes propres à la plateforme informatique.

La relation MOA et MOE est définie par un contrat qui précise leurs engagements mutuels.

***Le cycle de vie d'un logiciel*** désigne toutes les étapes du développement d'un logiciel, de sa conception jusqu’à sa disparition, le découpage en étapes permet de définir des jalons intermédiaires permettant la validation du développement logiciel par exemple :

Est-ce que le logiciel répond aux besoins exprimés ?

Détecter les erreurs au plus tôt possible car les erreurs détecter tardivement ont un impact très grave en terme de couts.

Le cycle de vie du logiciel comprend généralement au minimum les étapes suivantes :

**Analyse des besoins et faisabilité**

C’est-à-dire l'expression, le recueil et la formalisation des besoins du demandeur (le client) et de l'ensemble des contraintes, puis l'estimation de la faisabilité de ces besoins ;

**Spécifications ou conception générale**

Il s'agit de l'élaboration des spécifications de l'architecture générale du logiciel ;

**Conception détaillée**

cette étape consiste à définir précisément chaque sous-ensemble du logiciel ;

**Codage (Implémentation ou programmation)**

C’est la traduction dans un langage de programmation des fonctionnalités définies lors de phases de conception ;

**Tests unitaires**

Ils permettent de vérifier individuellement que chaque sous-ensemble du logiciel est implémenté conformément aux spécifications ;

**Intégration**

L’objectif est de s'assurer de l'interfaçage des différents éléments (modules) du logiciel. Elle fait l'objet de tests d'intégration consignés dans un document ;

**Qualification (ou recette)**

C’est-à-dire la vérification de la conformité du logiciel aux spécifications initiales ;

**Documentation**

Elle vise à produire les informations nécessaires pour l'utilisation du logiciel et pour des développements ultérieurs ;

**Mise en production**

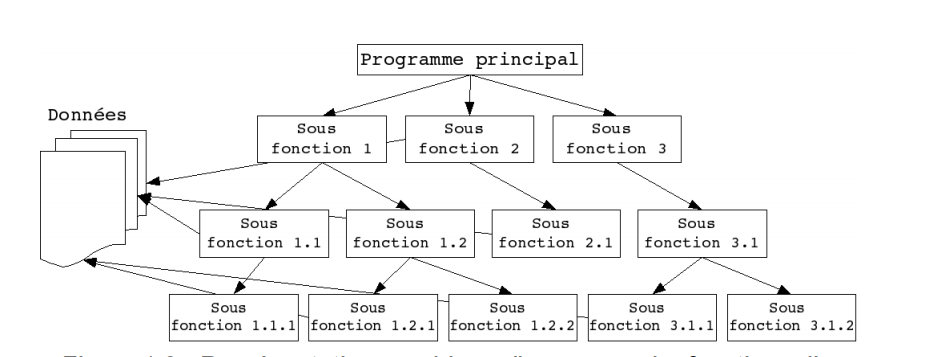
C’est le déploiement sur site du logiciel ;

**Maintenance**

Elle comprend toutes les actions correctives (maintenance corrective) et évolutives (maintenance évolutive) sur le logiciel.

***Méthodes d'analyse et de conception***

**Approche fonctionnelle :** Les méthodes fonctionnelles (également qualifiées de méthodes structurées) trouvent leur origine dans les langages procéduraux. Elles mettent en évidence les fonctions à assurer et proposent une approche hiérarchique descendante et modulaire.



La SADT (*Structured Analysis Design Technique*) est probablement la méthode d'analyse fonctionnelle et de gestion de projets la plus connue. Elle permet non seulement de décrire les tâches du projet et leurs interactions, mais aussi de décrire le système que le projet vise à étudier, créer ou modifier, en mettant notamment en évidence les parties qui constituent le système, la finalité et le fonctionnement de chacune, ainsi que les interfaces entre ces diverses parties. Le système ainsi modélisé n'est pas une simple collection d'éléments indépendants, mais une organisation structurée de ceux-ci dans une finalité précise

***L'approche orientée objet :*** considère qu'un système est un ensemble d'objets interagissant. Chaque objet dispose d'un ensemble d'attributs décrivant son état et l'état du système est décrit par l'état de l'ensemble, L'une des particularités de cette approche est qu'elle rapproche les données et leurs traitements (méthodes) associés au sein d'un unique objet. L'approche objet est une approche orientée donnée. Dans cette approche, les fonctions se déduisent d'un regroupement de champs de données formant une entité cohérente, logique, tangible et surtout stable quant au problème traité

Comme nous venons de le dire, un objet est caractérisé par plusieurs notions :

**L'identité**

L’objet possède une identité, qui permet de le distinguer des autres objets, indépendamment de son état. On construit généralement cette identité grâce à un identifiant découlant naturellement du problème (par exemple un produit pourra être repéré par un code, une voiture par un numéro de série, etc.) ;

**Les attributs**

Il s'agit des données caractérisant l'objet. Ce sont des variables stockant des informations sur l'état de l'objet ;

**Les méthodes**

Les méthodes d'un objet caractérisent son comportement, c'est-à-dire l'ensemble des actions (appelées opérations) que l'objet est à même de réaliser. Ces opérations permettent de faire réagir l'objet aux sollicitations extérieures (ou d'agir sur les autres objets). De plus, les opérations sont liées aux attributs, car leurs actions peuvent dépendre des valeurs des attributs, ou bien les modifier.

**Approche fonctionnelle vs approche objet**

En approche objet, l'évolution des besoins aura le plus souvent tendance à se présenter comme un changement de l'interaction des objets. S'il faut apporter une modification aux données, seul l'objet incriminé (encapsulant cette donnée) sera modifié. Toutes les fonctions à modifier sont bien identifiées : elles se trouvent dans ce même objet : ce sont ses méthodes. Dans une approche structurée, l'évolution des besoins entraîne souvent une dégénérescence, ou une profonde remise en question, car la décomposition des unités de traitement (du programme principal aux sous-fonctions) est directement dictée par ces besoins. D'autre part, une modification des données entraîne généralement une modification d'un nombre important de fonctions éparpillées (dispersées) et difficiles à identifier dans la hiérarchie de cette décomposition.

***Concepts importants de l'approche objet :***

***Une classe*** est un type de données abstrait qui précise des caractéristiques (attributs et méthodes) communes à toute une famille d'objets et qui permet de créer (instancier) des objets possédant ces caractéristiques.

***L'encapsulation*** consiste à masquer les détails d'implémentation d'un objet (attributs), tout en définissant une vue externe de l’objet, qui se base que sur les services accessibles (offerts) aux utilisateurs de l'objet. L'encapsulation donc garantit l'intégrité des données, car elle permet d'interdire, ou de restreindre, l'accès direct aux attributs des objets.

***L'héritage*** est un mécanisme de transmission des caractéristiques d'une classe (ses attributs et méthodes) vers une sous-classe. Une classe peut être spécialisée en d'autres classes, afin d'y ajouter des caractéristiques spécifiques ou d'en adapter certaines. Plusieurs classes peuvent être généralisées en une classe qui les factorise, afin de regrouper les caractéristiques communes d'un ensemble de classes. L'héritage évite la duplication et encourage la réutilisation, et également l’utilisation de polymorphisme.

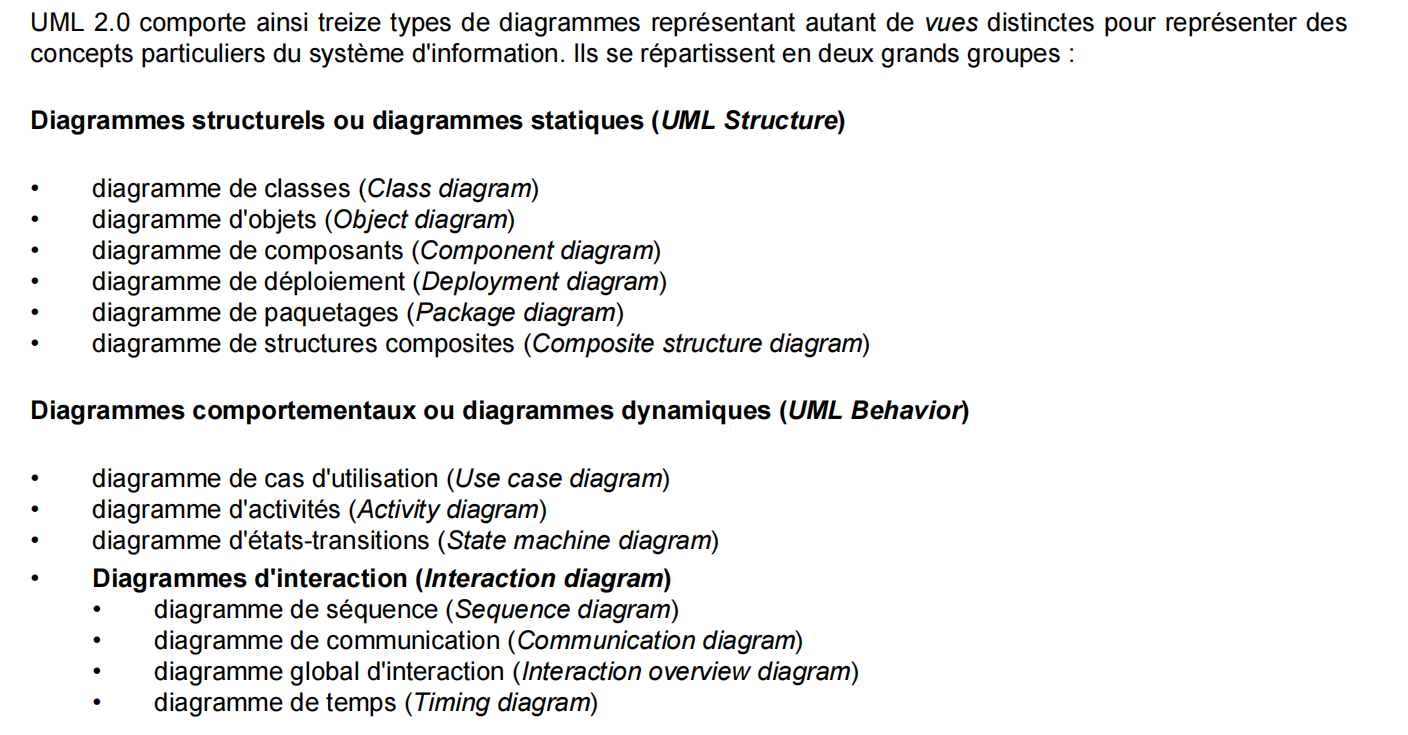
***Agrégation :*** Il s'agit d'une relation entre deux classes, spécifiant que les objets d'une classe sont des composants de l'autre classe. Une relation d'agrégation permet donc de définir des objets composés d'autres objets. L'agrégation permet donc d'assembler des objets de base, afin de construire des objets plus complexes.

***UML :*** langage de modélisation des systèmes d'information à objets, et c’est également un langage graphique qui permet de représenter et de communiquer les divers aspects d'un système d'information. Aux graphiques sont bien sûr associés des textes qui expliquent leur contenu. UML est donc un métalangage, car il fournit les éléments permettant de construire le modèle qui, lui, sera le langage du projet.

Pour programmer une application, il ne convient pas de se lancer tête baissée dans l'écriture du code : il faut d'abord organiser ses idées, les documenter, puis organiser la réalisation en définissant les modules et étapes de la réalisation. C'est cette démarche antérieure à l'écriture que l'on appelle ***modélisation* ;** son produit est un ***modèle.***

L'approche objet permet en principe à la maîtrise d'ouvrage de s'exprimer de façon précise selon un vocabulaire tout en transcrivant les besoins du métier qui pourra être immédiatement compris par les informaticiens.

Il est impossible de donner une représentation graphique complète d'un logiciel, Mais il est possible de donner sur un tel système des ***vues*** partielles, dont la conjonction donnera une idée utilisable en pratique sans risque d'erreur grave, pour cela :

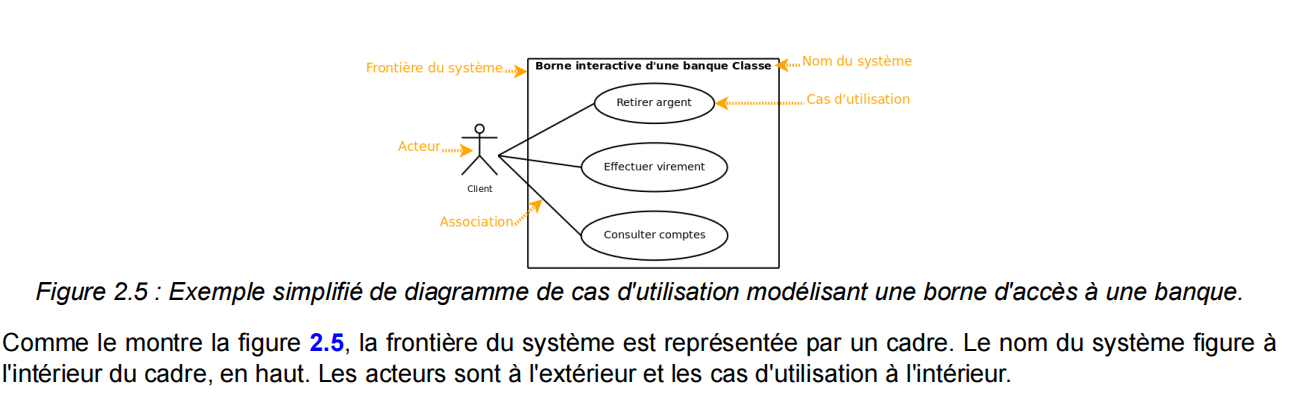


Ces diagrammes, d'une utilité variable selon les cas, ne sont pas nécessairement tous produits à l'occasion d'une modélisation. Les plus utiles pour la maîtrise d'ouvrage sont les diagrammes d'activités, de cas d'utilisation, de classes, d'objets, de séquence et d'états-transitions. Les diagrammes de composants, de déploiement et de communication sont surtout utiles pour la maîtrise d'œuvre à qui ils permettent de formaliser les contraintes de la réalisation et la solution technique.

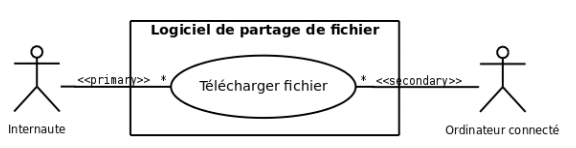
***Diagramme de cas d'utilisation :***  le diagramme de cas d'utilisation permet de recueillir (rassembler), d'analyser et d'organiser les besoins, et de recenser les grandes fonctionnalités d'un système, ainsi que de capturer le comportement d'un système tel qu’un utilisateur extérieur le voit, généralement il se compose d’un ensemble de cas d’utilisations et d’acteurs liée entre eux par des relations.

***Acteur :*** est l'idéalisation d'un rôle joué par une personne externe, un processus ou une chose qui interagit avec un système

***Un cas d'utilisation :*** modélise une fonctionnalité ou bien un service rendu par le système, sans imposer le mode de réalisation de ce service.



***Une relation d'association*** est chemin de communication entre un acteur et un cas d'utilisation représenté par un trait continu



***Acteurs principaux et secondaires :*** Un cas d'utilisation a au plus un acteur principal qui la déclenche tandis que l’acteur secondaire est sollicité pour des informations complémentaires.

***Cas d'utilisation interne :*** Quand un cas n'est pas directement relié à un acteur, il est qualifié de ***cas d'utilisation interne.***

***Relation d'inclusion << include >> :*** Un cas A inclut un cas B, c’est lorsque A est sollicité, B l'est obligatoirement, c’est comme si B une partie de A. Les inclusions permettent également de décomposer un cas complexe en sous-cas plus simples.

***Relation d'extension << extend >> :*** On dit qu'un cas d'utilisation A étend un cas d'utilisation B lorsque le cas d'utilisation A peut être appelé au cours de l'exécution du cas d'utilisation B. Exécuter B peut éventuellement entraîner l'exécution de A.

***Relations entre acteurs :*** La seule relation possible entre deux acteurs est la généralisation : un acteur A est une généralisation d'un acteur B si l'acteur A peut-être substitué par l'acteur B. Dans ce cas, tous les cas d'utilisation accessibles à A le sont aussi à B, mais l'inverse n'est pas vrai.

***Différence entre acteur et rôle :*** Chaque acteur doit être nommé. Ce nom doit refléter son rôle, plusieurs utilisateurs peuvent avoir le même rôle, et donc correspondre à un même acteur, et une même personne physique peut jouer des rôles différents vis-à-vis du système, et donc correspondre à plusieurs acteurs.

Imaginons les frontières d’un système. Tout ce qui est à l'extérieur et qui interagit avec le système est un acteur, tout ce qui est à l'intérieur est une fonctionnalité à réaliser.

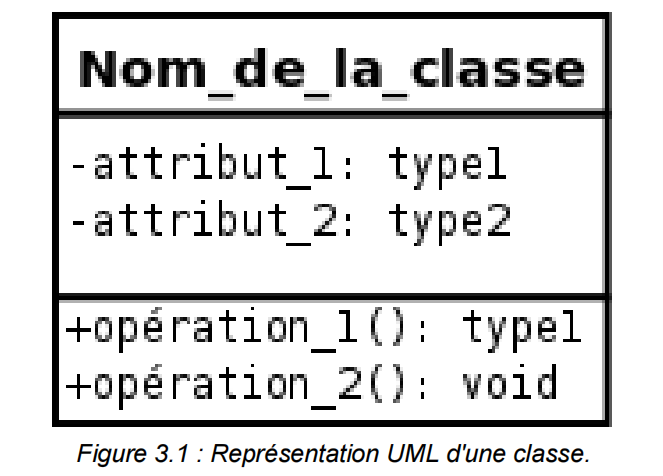
***Diagramme de classes :***  Il représente l'architecture conceptuelle du système : il décrit les classes que le système utilise, ainsi que leurs relations indépendamment d'un langage de programmation particulier.

***Instance*** est une concrétisation d'un concept abstrait qui est la classe.

***Une classe*** est la description formelle d'un ensemble d'objets ayant une sémantique et des caractéristiques (son *état* et son *comportement*) communes

***État d'un objet :*** ce sont les attributs et généralement les *terminaisons d'associations,* les attributs sont utilisés pour des valeurs de données pures, dépourvues d'identité, telles que les nombres et les chaînes de caractères.

***Comportement d'un objet :*** Ce sont des fonctions qui peuvent prendre des valeurs en entrée et modifier les attributs ou produire des résultats en sortie.



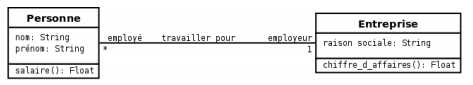
Encapsulation : est un mécanisme consistant á rassembler les données et les méthodes au sein d’une même structure en cachant l’implémentation de l’objet, c’est á dire en contrôlant l’accès au propriété par des méthodes getters and setters.

Attribut statique : est une propriété de la classe qui est partagé par toute les instances de la classe.

Méthode abstraite : c’est une méthode ou on connait sa déclaration, mais pas son implémentation.

Une classe abstraite pure : est une classe qui ne comporte que des méthodes abstraites, une telle classe est appelé une interface.

Association : est une relation entre 2 classes ou plus, qui indique les liens entre les instances de ses classes.



Il y’a l’association binaire, réflexive, n-aire.

Navigabilité : est une relation d’association dans un seul sens.

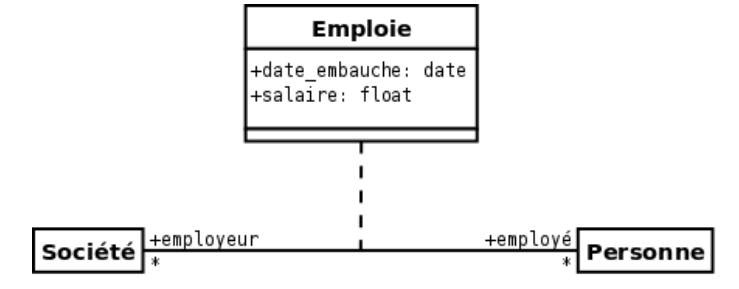


Ici chaque objet Commande contient une liste des produits, par contre produit ne comporte pas une liste des commandes.

Agrégation : est une relation de dépendance qui permet de définir des objets composés des autres, ou l’objet agrégé possède son propre cycle de vie, car la destruction de l’élément agrégat n’entraine pas la destruction de l’élément agrégé, ce traduit au niveau du langage C++ par l’utilisation des objets dynamiques.

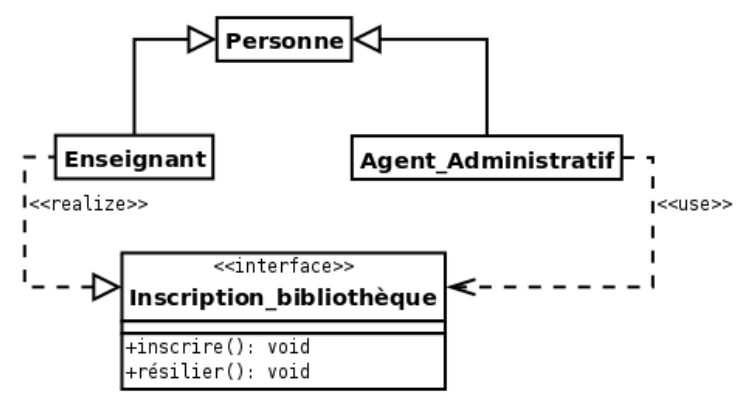
Composition : est une agrégation forte ou la destruction de l’élément composite entraine la destruction de l’élément composant, traduit en C++ par l’utilisation des objets statiques.

Classe d’association : est une association qui possède ces propres attributs, ces attributs dépend á la fois de 2 classes qui sont en relation.



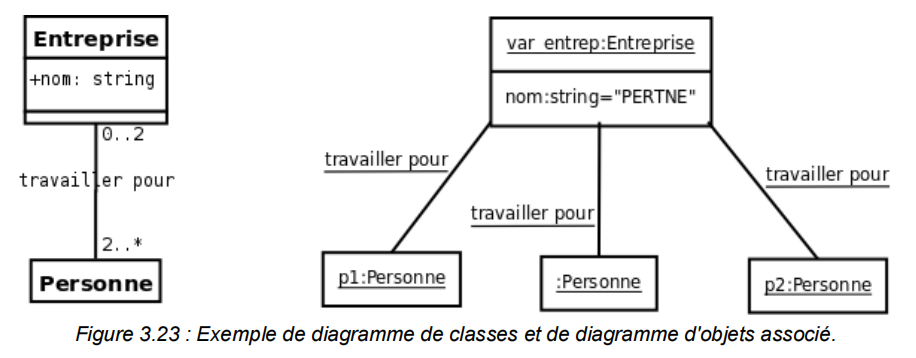
Généralisation : décrit une relation entre une classe générale et une classe spécialisé, ou la classe spécialiser considérer comme la classe générale mais qui comporte d’autre information supplémentaire.

Interface : une interface doit être implémenter par une ou plusieurs classe (relation realize), une classe qui utilise (dépend) de cette interface est représenter ça par le stéréotype <<use>>



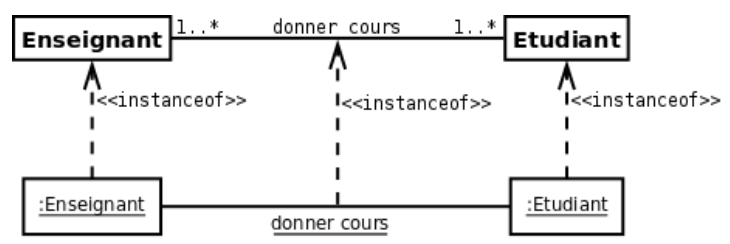
**Diagramme d’objet**

Un diagramme d’objets représente des objets et leur lien pour donner une vue figée de l’état system á une instante donnée, utilisé pour illustrer le diagramme de classes en montrant un exemple qui illustre le modèle.



Le nom de l’objet est souligné et précédé par << : >>, ces attributs possèdent des valeurs, les relations entre les classes deviennent des liens, généralisation ne possède pas d’instance, elle n’est plus donc représentée.

Relation de dépendance d’instanciation : décrit la relation entre une classe est ses instances.



OCL (Object Contraint Language) : il s’agit d’un langage formel pour l’expression des contraintes bien adaptés aux diagrammes d’UML, en particulier au diagramme de classes.

Context <élément> : pour indiquer l’élément sur lequel la contrainte s’applique.

inv : est une contrainte qui doit être respecter toujours par un objet ou un groupe d’objet.

pre : est une contrainte qui doit être vérifié avant l’appel á une opération, s’applique souvent sur les paramètres d’une méthodes.( par exemple pour tirer de l’argent il faut que le solde soit supérieur á 0).

post : est une contrainte qui doit être vérifier á la fin de l’exécution de l’opération.

init : permet de préciser la valeur initiale d’un attribut.

derive : La valeur de tels élément est déterminée en fonction des autres éléments.

On peut utiliser de teste if dans ocl :

if <expression\_logique\_0> then <expression\_logique\_1>

else <expression\_logique\_2> endif

self : utiliser pour faire référencier á un attribut ou une opération de l’objet désigné par le contexte.

oclIsTypeOf(t : oclType) : permet de vérifier que l’objet appelant appartient au type indiquer en paramètres.( Context = Société 🡪 self.oclIsType(Personne) renvoie false).

oclIsKindOf(t : oclType) : revoit true lorsque l’objet appelant est de même type ou un fils de la classe passé en paramètres.

oclIsNew() : utiliser dans une postcondition, indique que l’objet sur lequel elle est invoqué est créé pendant l’exécution de l’opération.

oclInState(State : s) : renvoie true si l’objet appartient á un des éléments de la liste passée en paramètres.

**«::»** permet de désigner un élément (comme une opération) dans un élément englobant (comme un classeur ou un paquetage) ;

**«.»** permet d'accéder à une caractéristique (attributs, terminaisons d'associations, opérations) d'un objet ;

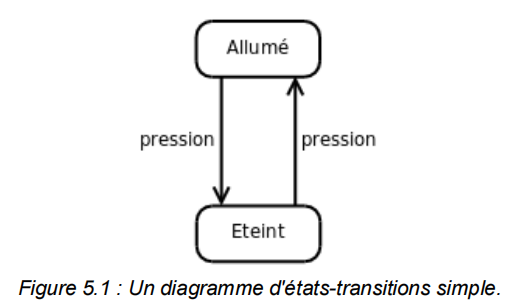
**«->»** permet d'accéder à une caractéristique d'une collection.

forAll : permet de vérifier c’est une expression logique est vrais sur tous les éléments d’une collection.

exists : permet de vérifie c’est une expression logique est vrais au moins pour un élément d’une collection.

**Diagramme d’état de transition**

Décrit le comportement interne d’un objet au cours de son cycle de vie á l’aide d’un automate á état finis.



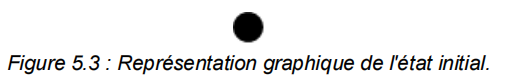
Un état est une période de vie d’un objet pendant laquelle ce dernier attend un évènement ou accomplit une activité.

Diagramme d’états de transition simple : (sans état de transition concurrente), un seul état actif á la fois.

État-globale : peut contenir plusieurs états actifs en même temps.

États concurrents : 2 états ou plus actifs á la fois.

États initiale : Lorsqu’un objet est créé, il entre dans l’état initiale.

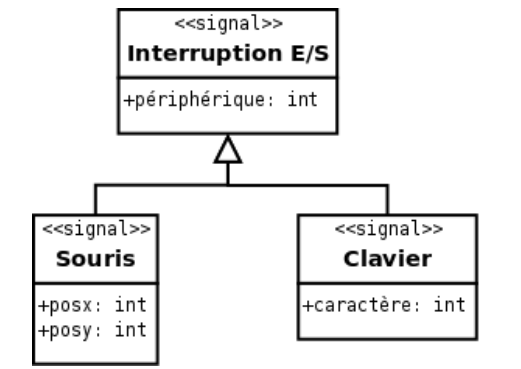


États final : la fin de l’objet.



Évènement : permet de transiter l’objet vers un nouvel état, il y’a les évènements explicites et implicites : signal, appel, changement, temporel.

Évènement de type signal : la réception du signal est un évènement pour l’objet destinataire, signal est un classeur qui ne fournis pas d’opérations et dont les attributs sont interprétés par des arguments, destiné á véhiculer une communication asynchrone á sens unique entre 2 objets. L’objet expéditeur envoie une instance de signale vers un ou plusieurs objets destinataires.



Les signaux la relation de généralisation.

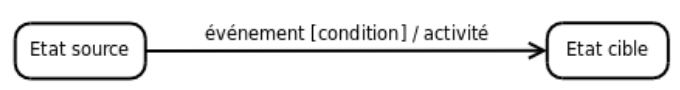
Évènement appel : est la réception de l’appel d’une opération par un objet.

Événement de changement : c’est l’évènement de changement des valeurs d’attributs d’un objet par la satisfaction d’une expression boolean (vérification d’un teste).

Événement temporel : des évènements qui sont gérée par le passage du temps.

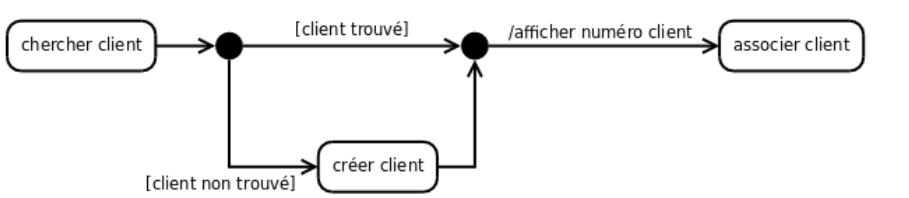
**Transition :** est la réponse d’un objet á l’occurrence d’un évènement, le passage d’un état 1 á état 2.

Transition externe : une transition qui change l’état actif d’un objet.



Transition d’achèvement : est une transition qui n’a pas d’événement déclencheur, contient une condition de garde qu’elle la déclenche.

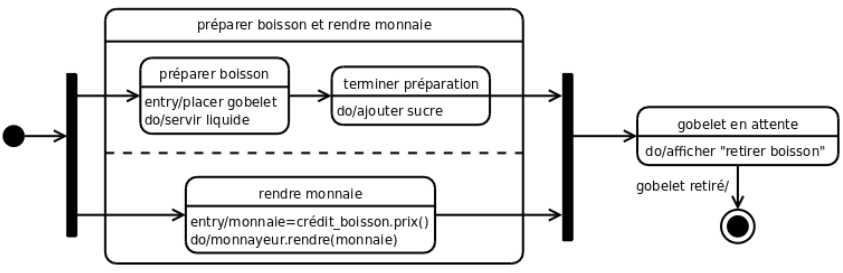
Transition interne : ??.

Les points de jonctions : permet d’organisé les transitions en des segments de transitions entrantes et sortantes, et de représenter des alternatifs qui rend le diagramme plus lisible`

Point de décision: possédé une entrée et au moins deux sorties, utiliser pour présenter des choix multiple, comporte aussi la garde [else].

État composite : un état composite est un état composé de plusieurs sous-états.

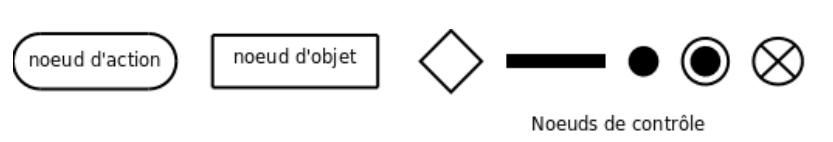
État orthogonale : est un état composite contient plus d’une région qui représente un flot d’exécution (différent états concurrents), séparer par des traits horizontaux en pointillé, utiliser pour représenter les états concurrents.



Les transitions complexes : représenté par une barre épaisse, utiliser pour faire apparaitre la notion de concurrence.

**Diagramme d’activité**

Permet de représenter graphiquement le comportement d’une méthode ou le déroulement d’un cas d’utilisation á l’aide d’un automate fini qui comporte un ensemble des nœuds reliant entre eux qui construit le diagramme d’activité.

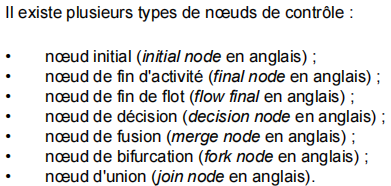


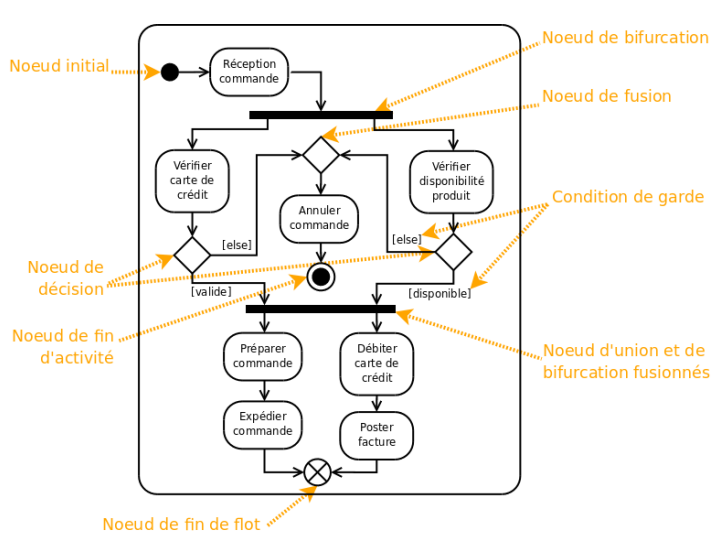
Action : ce sont des étapes discrètes á partir desquelles se construisent les comportements, instruction élémentaire dans un langage de programmation.

Activité : est un comportement décrit par une séquence organisé d’action, le passage d’une activité á une autre est matérialisé par une transition.

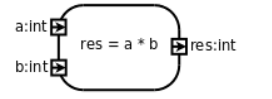
Nœud d’action : est un nœud d’activité exécutable représente une transformation ou un calcul dans le système modélisé, généralement liées á des opérations qui sont directement invoqué.

Nœud de contrôle : utilisé pour coordonner les flots entre les nœud d’une activité.

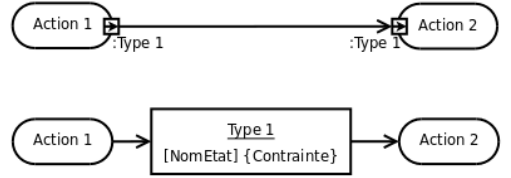


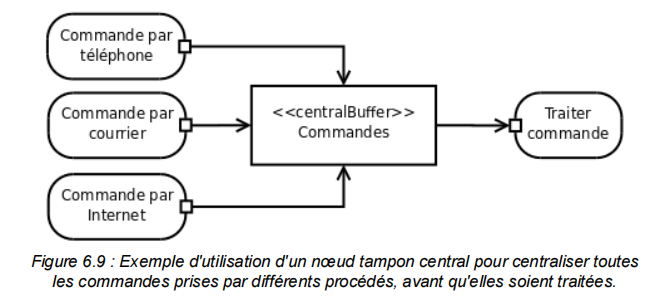


Nœud d’objet : utiliser pour représenter un objet dans le diagramme d’activité, se constitues des pins d’entrées sorties, l’activité commence lorsque en affectant des valeurs á ces entrées, et se termine lorsqu’une valeur est disponible á sa sortie.

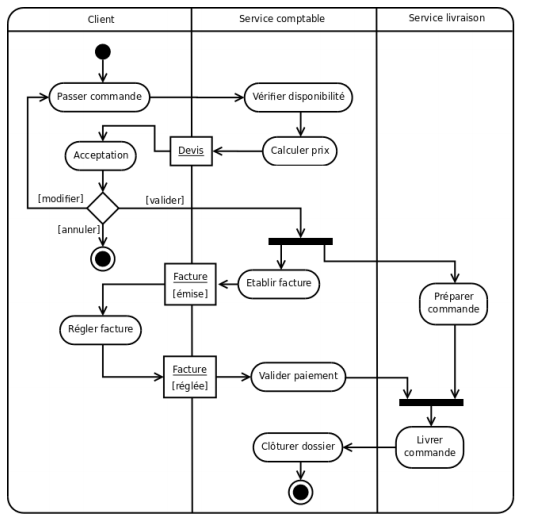


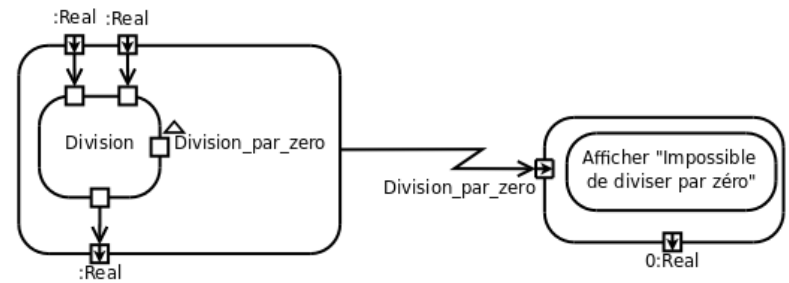
Un flot d’objet permet de passer des valeurs d’une activité á une autre.



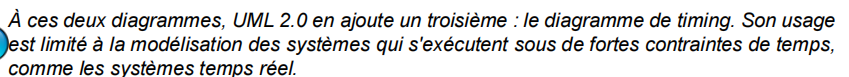


Partition : ce sont des couloirs permettent d’organiser les nœuds d’activités dans des regroupements de spécifications bien déterminer.



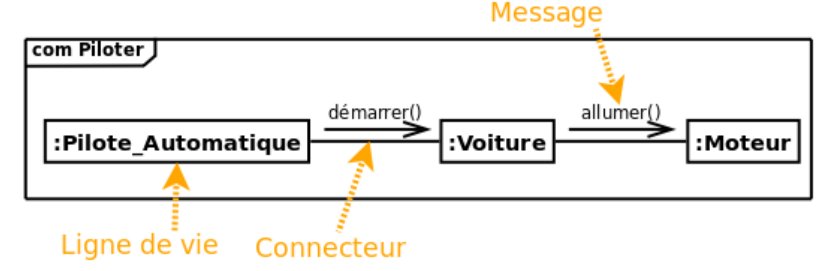


**Diagramme d’interaction :** établir le lien entre le diagramme de cas d’utilisation et le diagramme de classes, il montre comment les objets communiquent pour réaliser une certaine fonctionnalité. Ensemble d’éléments interagissent entre eux pour décrire un comportement particulier, pour ça on utilise soit diagramme de communication soit diagramme de séquence.

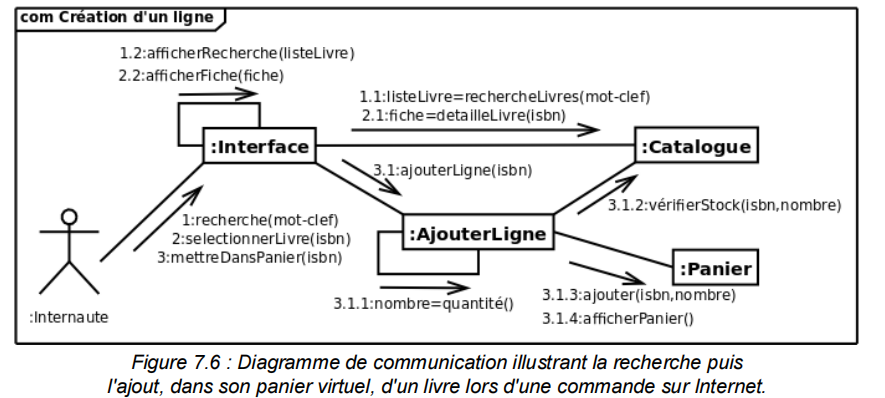


**Diagramme de communication**

Mettre l’acescent sur l’organisation spatiale des objets qui envoient et reçoivent des messages, utiliser pour illustre un cas d’utilisation ou pour décrire une opération.

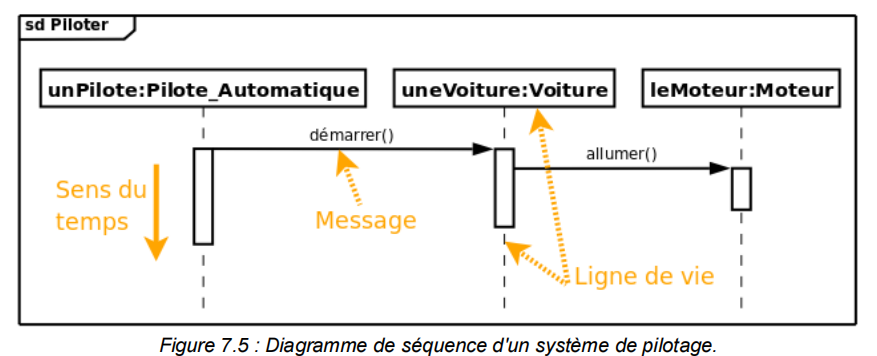


Les messages sont généralement ordonnés selon un numéro de séquence croissant.



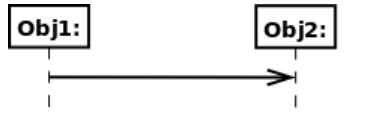
**Diagramme de séquence**

Comme un diagramme de communication, mais avec la représentation du temps par une dimension qui se découle de haute en bas.

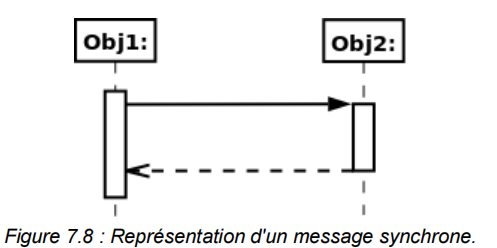


Type de messages :

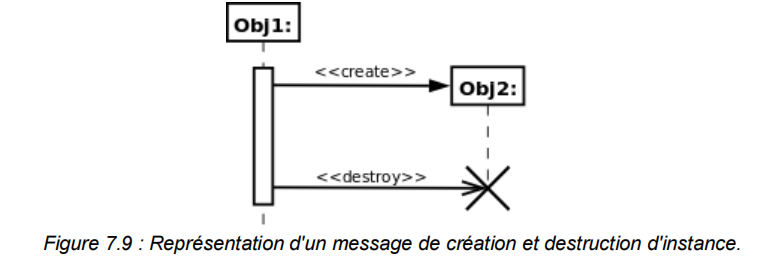
Message asynchrone : l’émetteur ne reste pas bloqué en attendant que le récepteur lui répond.



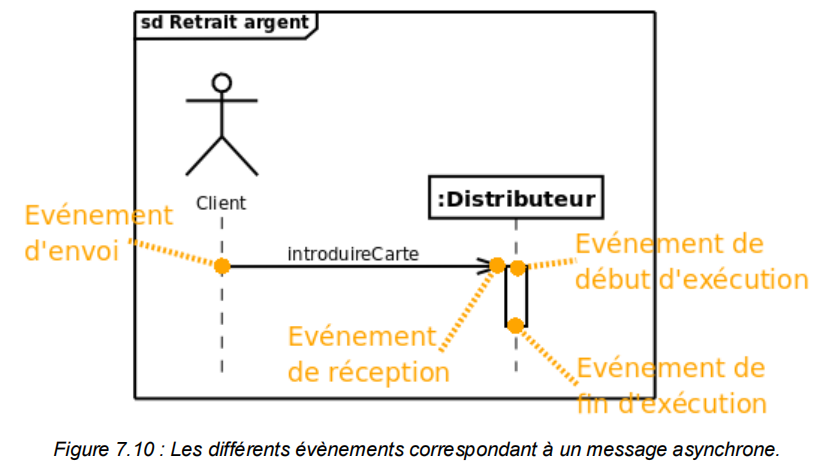
Message synchrone : l’émetteur reste bloqué jusqu’à ce qu’il reçoit un message de retour.



Message de création et destruction d’une instance :



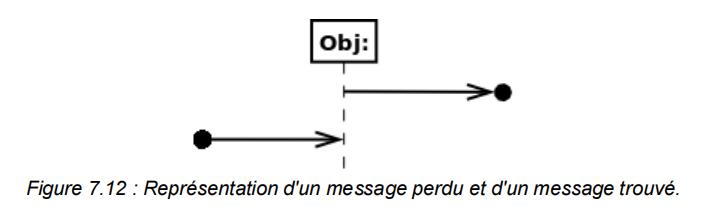
Évènement et messages : un message peut se constituer de plusieurs évènements.



La plupart des cas, la réception d’un message est suivie par l’exécution d’une méthodes d’une classe

Message found : un message found c’est lorsque l’évènement de d’émission n’est pas connu.

Message not found : un message not found c’est lorsque l’évènement de réception n’est pas connu.



Les fragments :

Alt : équivalent á une exécution á choix multiple.

Opt : comporte une condition de garde qui indique que le fragment sera exécuté si la condition de garde est satisfaite.

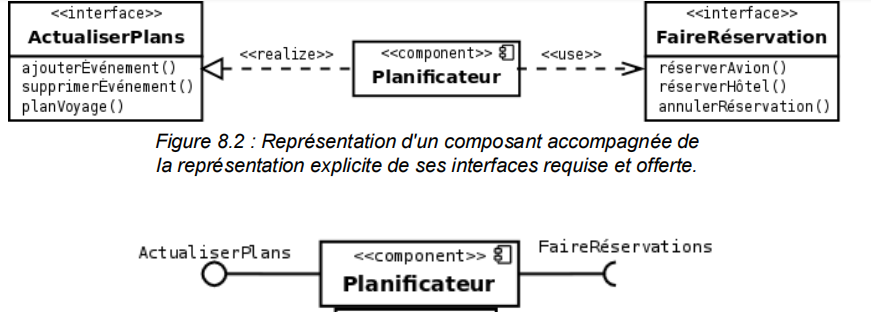
Loop : utiliser pour exécuter un fragment plusieurs fois entre une valeur min et une valeur max, comporte aussi une condition de garde.

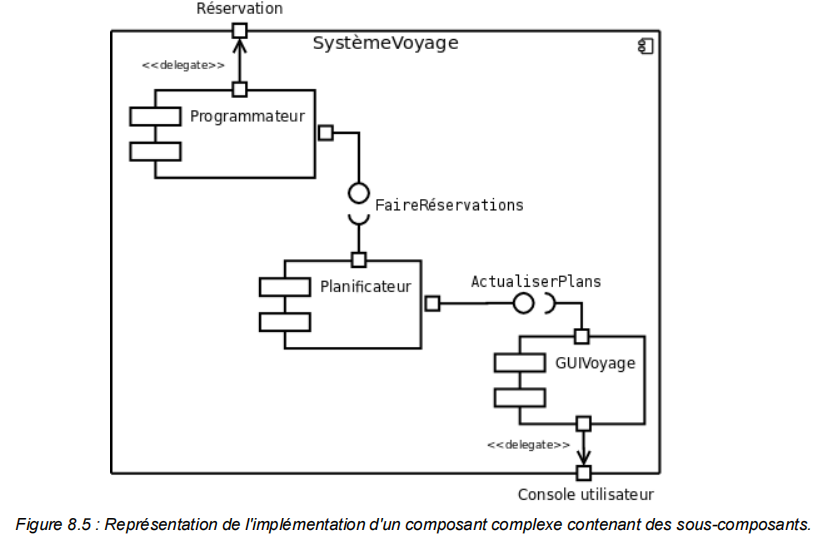
Par : pour exécuter plusieurs fragments en parallèle.

Strict : comporte au moins 2 fragments, mais qui s’exécute selon leur ordre d’apparition (l’ordre n’est pas connu au préalable).

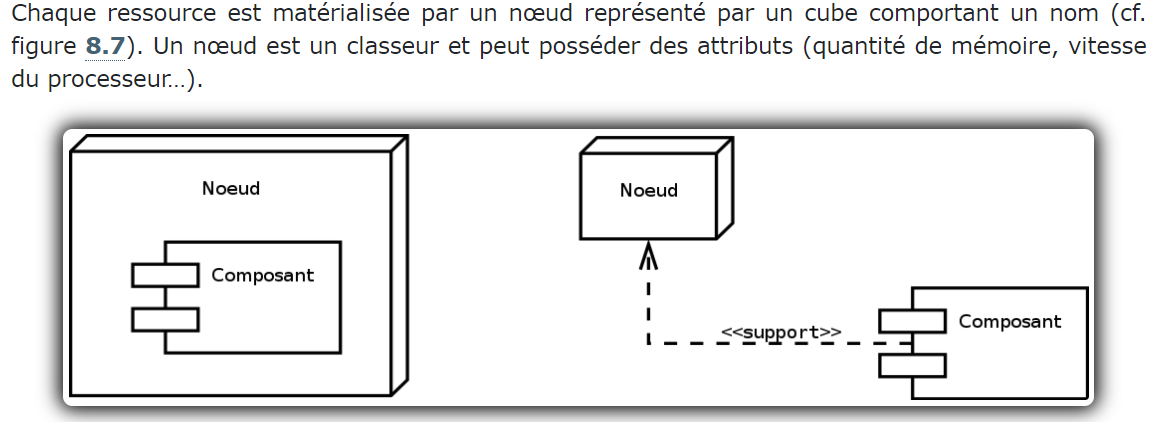
**Diagramme de composant :** décrit le système modélisé sous forme des composants réutilisables et leurs relations de dépendance qui peut être soit réalisation soit utilisation.

Un composant doit fournir un service bien précis, comporte une ou plusieurs interface requises et offertes, son comportement interne est totalement masqué (ensemble des classes), seul ses interfaces qui sont visible, et pour changer un composant par un autre il faut respecter les interfaces requises et offertes.





**Diagramme de déploiement** : utiliser pour représenter le déploiement des composant sur les ressources matérielles.



**Diagramme de package :** un diagramme de package permet de regrouper des classes, des classes, des interfaces, des diagrammes …, Ils fournissent une perspective de la structure hiérarchique des différents éléments UML au sein d'un système donné.

