Formulaire UML

Sommaire

I. *Les concepts de l'approche Objet*

1. Introduction

Dans un langage orienté objet, on ne manipule que des objets. Un objet est une modélisation soit d'une réalité vivante (être humain, animal), soit d'une réalité matérielle (voiture, bouteille) soit d'une réalité immatérielle ou abstraite (idée, un compte bancaire …).
Parmi les caractéristiques d'un objet, on distingue ces qualités propres (attributs) qui sont une description de l'objet, et ces capacités (méthodes) qui sont l'ensemble des actions que l'on peut entreprendre avec cet objet. Ce type de langage se distingue donc complètement des langages dits procéduraux comme ADA.
En plus de la description des objets, il convient, comme nous le verrons plus tard, de définir les relations entre ces objets et comment ils interagissent entre eux.

2. Description d'un objet

Un objet est divisé en 3 parties :

- L'identité de l'objet
  C'est le nom que l'on donne à cet objet, ce qui permet de le distinguer d'un autre objet du même type.
- L'état de l'objet
  C'est l'ensemble des valeurs des attributs qui décrivent cet objet.
- Le comportement de l'objet
  C'est l'ensemble des méthodes qui décrivent les actions que notre objet peut entreprendre.

3. Etat + Comportement

Comme dit précédemment, ce sont les valeurs des attributs qui déterminent l'état d'un objet. Les attributs d'un objet ne peuvent pas prendre n'importe quelles valeurs : un objet doit toujours être dans un état cohérent (c'est-à-dire un état prévu dans les spécifications). Par exemple, si on considère un

attribut qui décrit une quantité de liquide, il est impossible d'avoir une quantité négative. Pour que l'objet soit dans un état cohérent, il faut que cet attribut ait toujours une valeur positive.
Ainsi le comportement d'un objet dépend de l'état dans lequel il se trouve, et une opération donnée ne s'exécutera pas de la même manière pour deux états différents de l'objet. Si on reprend l'exemple précédent, et que l'on suppose qu'il existe une méthode qui permet d'enlever une unité de liquide, alors cette méthode ne fera rien si la quantité de liquide est déjà nulle. Ainsi, on assure l'intégrité/la cohérence de l'état de notre objet.

4. L'encapsulation

Ce second concept de la programmation orientée objet découle directement du précédent. Nous avons vu l'utilité du principe de cohérence d'un objet. Pour assurer celui-ci, et puisque l'exécution d'une méthode assure l'intégrité de l'objet, l'état d'un objet ne peut être modifié que par une méthode. Ainsi, l'utilisateur d'une classe ne peut pas manipuler directement les attributs de cette classe, il doit obligatoirement utiliser les méthodes, et la cohérence d'un objet est assurée.

5. L'héritage

La généralisation ou l'héritage est une relation entre deux classes qui indique que l'une est une spécialisation de l'autre. Si on considère les deux classes félin et chat. Le chat est une sorte particulière de félin. On appelle la classe la plus générale 'mère' et la classe spécialisée 'fille'.
Voici la caractéristique de la classe fille :

- Elle possède les mêmes descriptions que la classe mère (attributs et méthodes)

- Elle possède en plus des descriptions qui lui sont propres (attributs et/ou méthodes). On peut tout à fait modifier l'implémentation d'une méthode définie dans la classe mère (on parle de surcharge).

6. Le polymorphisme

Certaines classes peuvent ne pas être instanciées (c'est-à-dire qu'il n'existe pas d'objets de ce type). On dit qu'elles sont abstraites.
Elles ne seront utilisées qu'en tant que classes mères pour des héritages. Pour reconnaitre une classe abstraite, on peut se souvenir que puisqu'elles ne sont pas 'instanciables', une de leurs méthodes au moins n'est pas implémentée (on parle de méthode abstraite dont on donne juste la signature et qui sera implémentée dans les classes filles).
Les différentes classes filles qui hériteront d'une même classe mère abstraite, auront une partie de leur comportement en commun : c'est ce qu'on appelle le polymorphisme.

II. Diagramme de Classe

1. La dépendance (dependency)

La dépendance est le lien le plus pauvre qui existe entre deux classes. Elle traduit le fin qu'à un moment ou à un autre la classe A dépend de la classe B car A utilise B. En général, il s'agit d'une interaction temporaire entre les deux classes (typiquement, A va utiliser les services de B) ; une modification de l'élément dont on dépend peut nécessiter une mise à jour de l'élément dépendant.

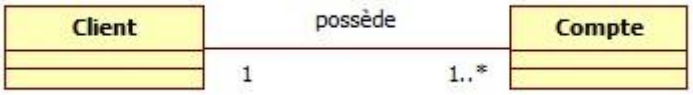
Exemple

---

2. L'association (association)

L'association est un lien plus fort que la dépendance. Une association exprime une connexion sémantique bidirectionnelle entre les deux classes. C'est-à-dire que l'on peut qualifier l'association qui relie les deux classes (contrairement à la dépendance). Attention, deux classes associées ont une 'vie' indépendante.
Si l'on peut dire que A a un B alors il est probable que l'on est à faire à une association.

Exemple



3. L'agrégation (aggregation)

L'agrégation est une forme contrainte d'association dans laquelle l'une des classes décrit un tout alors que la classe associée décrit une ou des parties. On appelle le tout composé et une partie un composant.
On va utiliser l'agrégation lorsqu'une classe B est composée d'objets de la classe A. Attention les objets de la classe A sont indépendants dans le sens où ils peuvent exister sans B.

Exemple

4. La composition (composition)

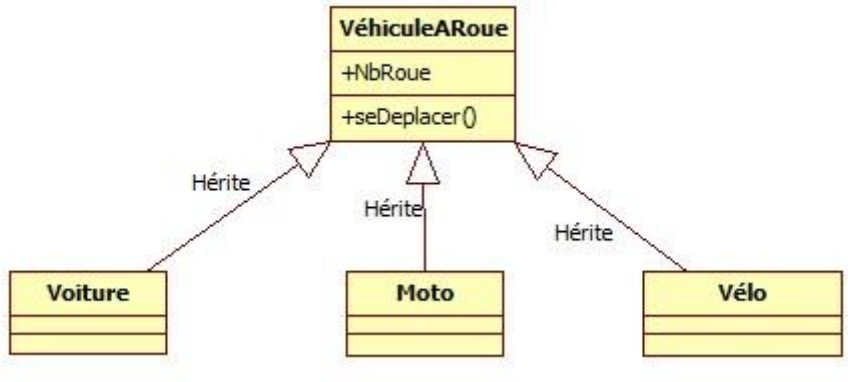
Une composition est une forme contrainte d'agrégation. Dans le cas de l'agrégation, les objets de la classe A sont liés à celui de la classe B, et ils ont une durée de vie strictement incluse dans celle de leur composée. Il ne peut pas exister de A sans B.

Exemple

5. L'héritage (generalization)

La relation d'héritage renvoie directement au concept d'héritage présenté dans la première partie. Une classe A hérite d'une classe B si c'est une classe particulière ou une sous-classe de B (on parle de catégorisation).

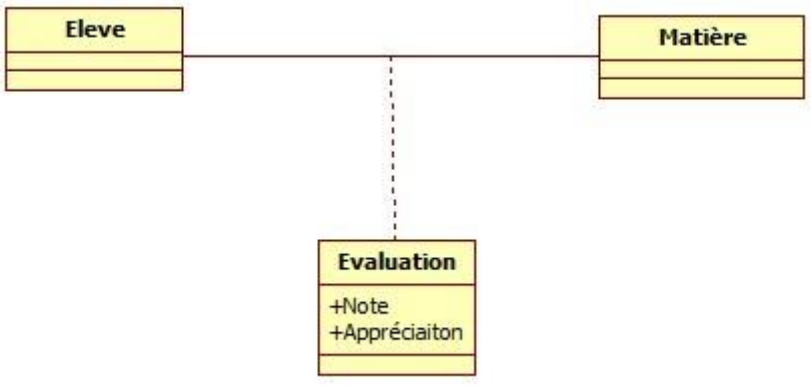
Exemple



6. L'association de classe (association class)

Il peut arriver qu'une association entre deux classes ne soit pas une simple structure de connexion. Lorsque l'association est porteuse d'informations, on utilise une classe d'association qui possède un nom, des attributs …

Exemple



7. Un cas particulier : l'énumération

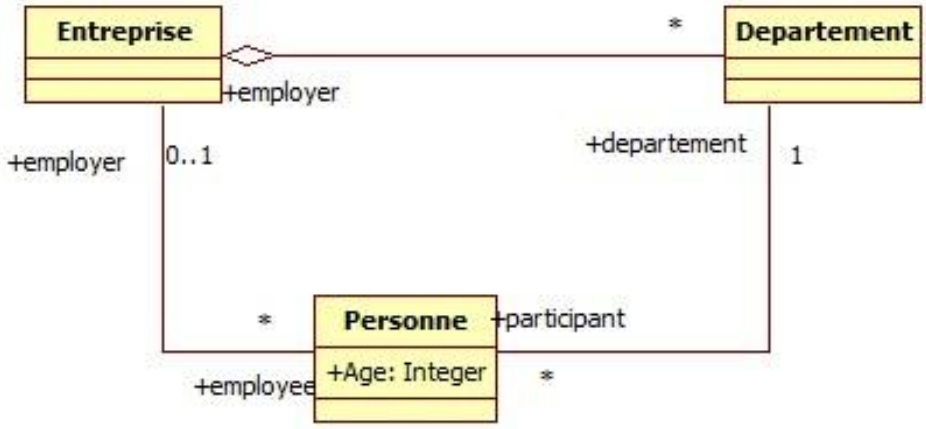
Une énumération est un type de classe particulier qui représente un objet qui ne peut prendre qu'un nombre fini de valeurs.

III. Les contraintes OCL

L'UML ne permettant pas de tout décrire, les développeurs ont eu besoin de développer un langage qui permet de décrire les contraintes qu'il peut exister dans un diagramme de classe : l'OCL. C'est un langage déclaratif qui décrit le quoi (et non le comment).

1. Un peu de syntaxe

Une contrainte OCL est toujours écrite entre crochets. Il s'agit d'une expression logique qui doit toujours être vérifiée. On doit toujours déclarer le contexte (c'est-à-dire la classe avec laquelle on travaille), puis le type de contrainte que l'on va décrire : il en existe de trois types, à savoir les invariants, les pré-conditions et les post-conditions. De manière générale, pour accéder au contexte on utilise le mot clé 'self', et pour accéder à un attribut « att » d'un objet « o » on utilise la ponction suivante : « o.att ». On utilise aussi le «.» pour accéder à une autre classe à laquelle on est associée (on réfère au nom qui se situe à la fin de l'association).
Voici quelques exemples pour mieux comprendre.

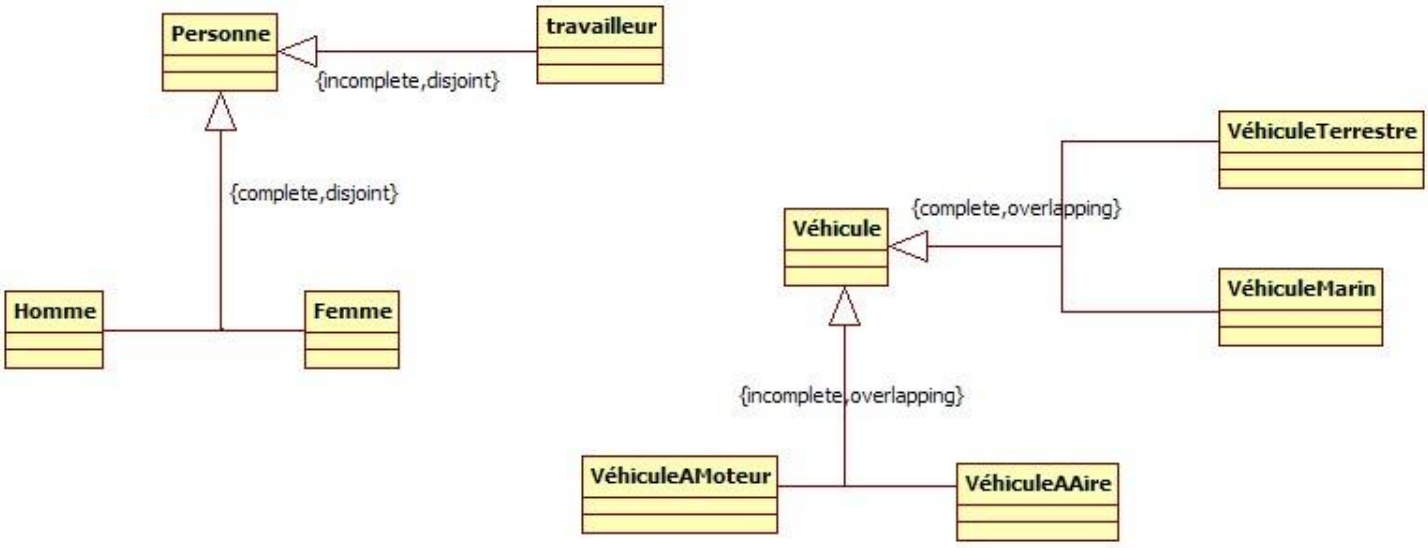



```
{context Personne
inv : self.employer = self.departement.employer
inv : self.age > 0
}
```

2. Les contraintes de généralisation

- {complete, disjoint} : recouvre complètement la classe mère, et pas d'instance commune.
- {incomplete, disjoint} : ne recouvre pas la classe mère, et pas d'instance commune.
- {complete, overlapping} : recouvre complètement la classe mère, et peut avoir des instances communes.
- {incomplete, overlapping} : ne recouvre pas la classe mère, et peut avoir des instances communes.

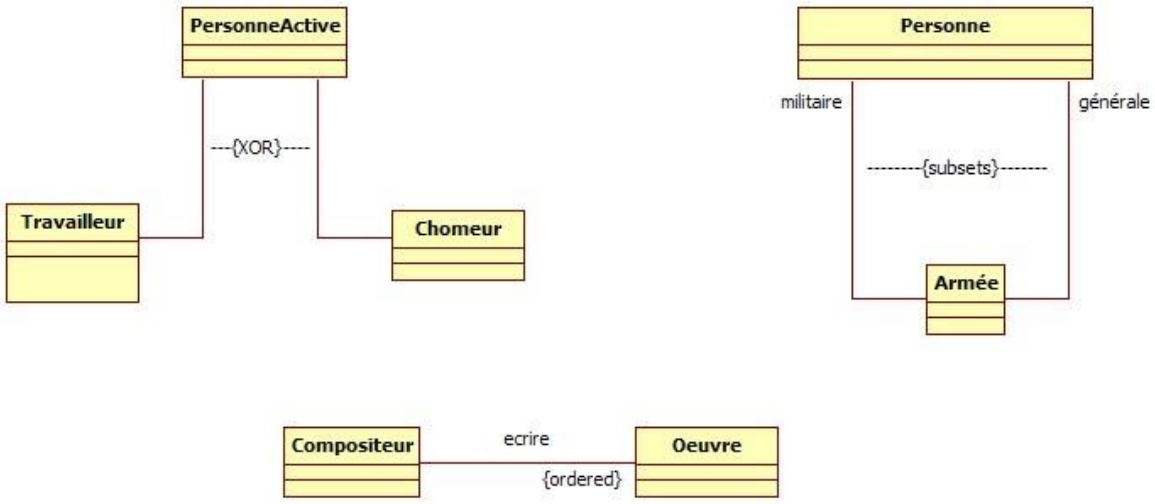
Voici un exemple qui récapitule toutes ces notions :



3. Les contraintes d'association

- {subsets <property name>} : il s'agit d'une sous-partie (oriéntée)
- {redefines <property name>} :
- {OR}
- {XOR}
- {union} :
- {ordered} : les éléments ne peuvent survenir qu'une seule fois et l'ordre est important.
- {bag} : les éléments peuvent survenir plusieurs fois et l'ordre n'a pas d'importance.
- {sequence} : les éléments peuvent survenir plusieurs fois et l'ordre est important.

Voici quelques exemples pour comprendre :



4. Les contraintes sur les opérations

Nous avons vu précédemment la syntaxe pour déclarer une contrainte sur une classe. Pour définir des contraintes sur une opération, la syntaxe est différente. Tout d'abord on précise le contexte comme suit (la syntaxe est la même si le contexte est un attribut ou l'autre bout d'une association) :
context classeA :: methode1(arg1 : int) : int
De plus, pour une opération, on distingue la pré-condition (état avant l'exécution) et la post-condition (état après l'exécution). Lors de la post-condition, on peut accéder à deux éléments spécifiques supplémentaires : result qui réfère à la valeur de retour et nomAttribut@pre qui réfère à la valeur de l'attribut avant l'appel de la fonction.
Deux petits exemples :

{context Compte :: getSolde() : int
Post : result = self.solde
}
{context Compte :: crediter(montant : int)
Pre : montant > 0
Post : self.solde = self.solde@pre + montant
}

On peut aussi donner un nom à une contrainte (on le met entre le type de la contrainte et les deux points) et faire des commentaires de ligne (--).
Il existe encore deux autres types de contrainte (rare) : « init » qui permet d'initialiser un attribut et « derive » lorsque un attribut est déduit d'un ou d'autres attribut(s).

•**context**TypeName::AttributeName: Type
init: --some expression representing the initial value
•**context**TypeName::AttributeName: Type
derive:--some expression representing the derivation rule

5. Les contraintes sur les ensembles

Une collection est la super-classe abstraite de Set,OrderedSet, Bag et Sequence.

- Set est un ensemble sans répétition. Set n'est pas ordonné.
- OrderedSet est un ensemble sans répétition. OrderedSet est ordonné.
- Bag est un ensemble avec répétition. Bag n'est pas ordonné.
- Sequence est un ensemble avec répétition. Sequence est ordonné.

Voici la liste des opérations communes à tous les ensembles :

- Le nombre d'élément dans la collection elle-même : size()
- Retourne vrai si l'objet est dans la collection : includes(objet)
- Retourne vrai si tous les objets sont dans la collection: includesAll(collection)
- Retourne vrai si l'objet n'est pas dans la collection: excludes(objet)
- Le nombre d'occurrences d'un objet dans la collection : count()
- L'absence d'une sous-collection dans la collection: excludesAll(collection)
- Le fait que la collection est vide: isEmpty()
- Le fait que la collection n'est pas vide: notEmpty()
- Retourne une sous-collection d'éléments qui vérifient l'expression: select(expression)
- Retourne une sous-collection d'éléments qui ne vérifient pas l'expression: reject(expression)
- Collecte tous les éléments qui vérifient l'expression et retourne ceux-ci dans une nouvelle collection: collect(expression)
- Retourne au hasard un élément qui satisfait l'expression : any(expression)
- Retourne vrai si l'expression est vraie au moins une fois: exists(expression)
- Retourne vrai si l'expression est vraie exactement une fois: one(expression)
- Retourne vrai si l'expression est tout le temps vraie: forAll(expression)
- Retourne la somme de tous les elements (il faut qu'ils soient d'un type numérique) : sum()

Pour utiliser ces différentes opérations, on utilise la notation suivante : '->'.

{context Person p
Inv : p.child->size() >= 0
}

ATTENTION, la multiplicité 0..1 est considérée comme une collection.

6. Les contraintes conditionnelles

If exp1 then exp2 else exp3 endif : si l'expression 1 est vérifiée alors l'expression2 est vraie, sinon l'expression3 est vraie.

exp1 implies exp2 : si l'expression1 est vraie alors l'expression2 doit être vraie.

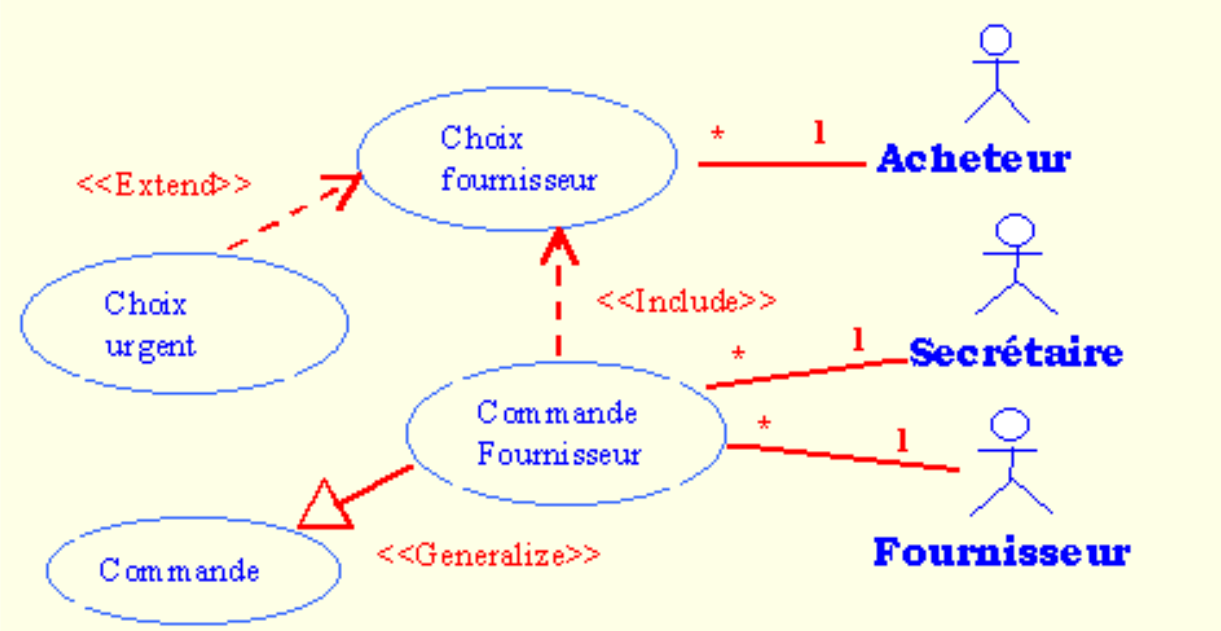
IV. *Les cas d'utilisation (Use Case)*

Le cas d'utilisation permet de décrire un besoin fonctionnel du système comme un tout d'un point de vue externe. Il s'agit en fait des différentes utilisations que l'on peut faire du système.
Pour décrire un cas d'utilisation, il faut identifier : les acteurs (type d'utilisateur qui interagissent avec le système), le système et la raison pour laquelle l'acteur utilise le système.
Un acteur est externe au système. Un acteur décrit un rôle que peut jouer l'utilisateur en relation avec le système. Un acteur n'est pas nécessairement une personne, ça peut être un autre système.
Une relation entre le système et un utilisateur peut être de trois sortes :

- L'acteur actionne le use case
- Le use case retourne un résultat à l'acteur
- Les deux …

Il peut exister des relations entre les différents use case :

- A Includes B : signifie que l'exécution du use case A inclus toujours l'exécution du use case B. Cependant, le use case B peut être exécuté par d'autres use case.
- A Extends B : A extend B signifie que le comportement d'un B peut être complété par le comportement d'un A. La relation extend doit spécifier à la fois : la condition de l'extension (par exemple RuptureStock:=Vrai) et le point d'extension, i.e. l'endroit où l'extension doit être faite dans le Use Case général
- A is a Generalization of B : équivalent à B hérite de A.

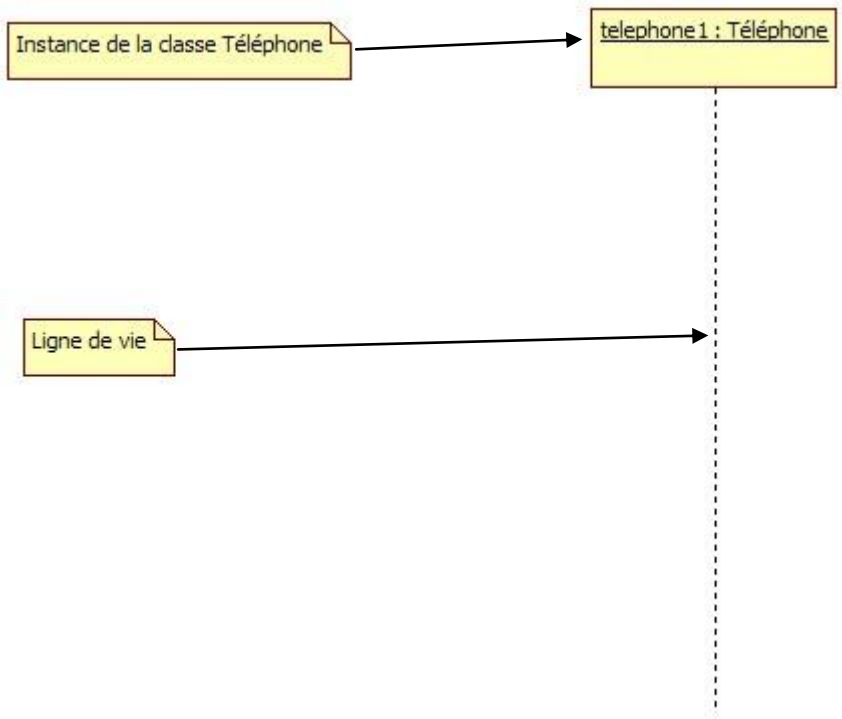


V. *Les diagrammes de séquence*

Un diagramme de séquence permet de décrire la structure dynamique d'un système. Il s'agit de représenter une séquence d'évènements (point de vue temporel) en utilisant les objets impliqués et en explicitant les interactions entre ces objets.
Pour chaque scénario défini dans les cas d'utilisation, on doit décrire le diagramme de séquence qui lui est associé :

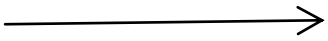
Diagramme séquence ⇔ scénario

Attention, dans un diagramme de séquence, on utilise des instances des objets, ils sont donc représentés avec une ligne de vie :



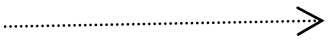
Un diagramme de séquence montre comment les objets impliqués dans un même scénario interagissent entre eux. Le seul moyen pour un objet A d'interagir avec un objet B est de lui envoyer un message. L'envoie de message est représenté par une flèche depuis l'émetteur vers le destinataire et le contenu du message est écrit au dessus de la flèche.
Il existe différentes formes de flèches selon le type de message :

- ✓ Fine ligne avec flèche ouverte : message asynchrone, signifie que l'émetteur continue sa liste d'instruction après l'envoie.

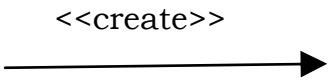


- ✓ Fine ligne avec flèche remplie : message synchrone, signifie que l'émetteur attend un message de retour avec de continuer sa liste d'instructions. Cela correspond à l'appel de méthodes.

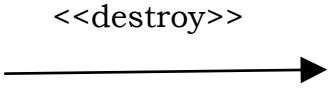
- ✓ Ligne en pointillée avec flèche ouverte : message de retour.



- ✓ Fine ligne avec flèche remplie surmontée de <<create>> : création d'un objet.

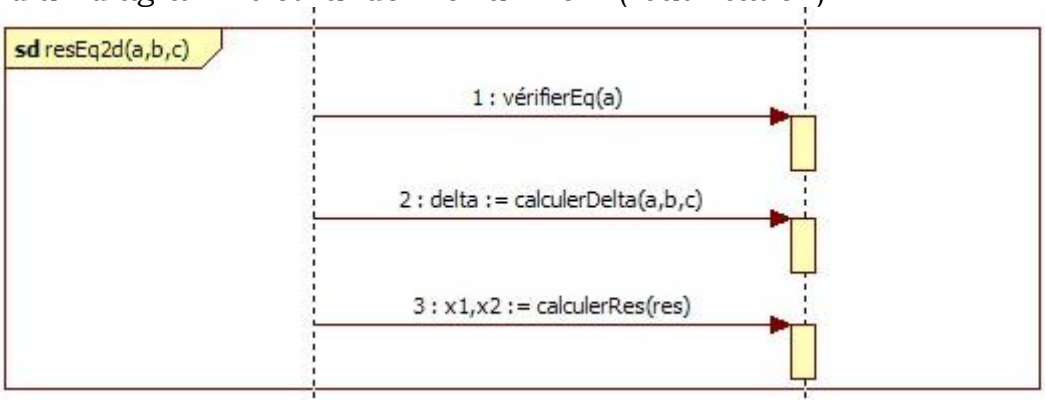


- ✓ Fine ligne avec flèche remplie surmontée de <<destroy>> : destruction d'un objet.

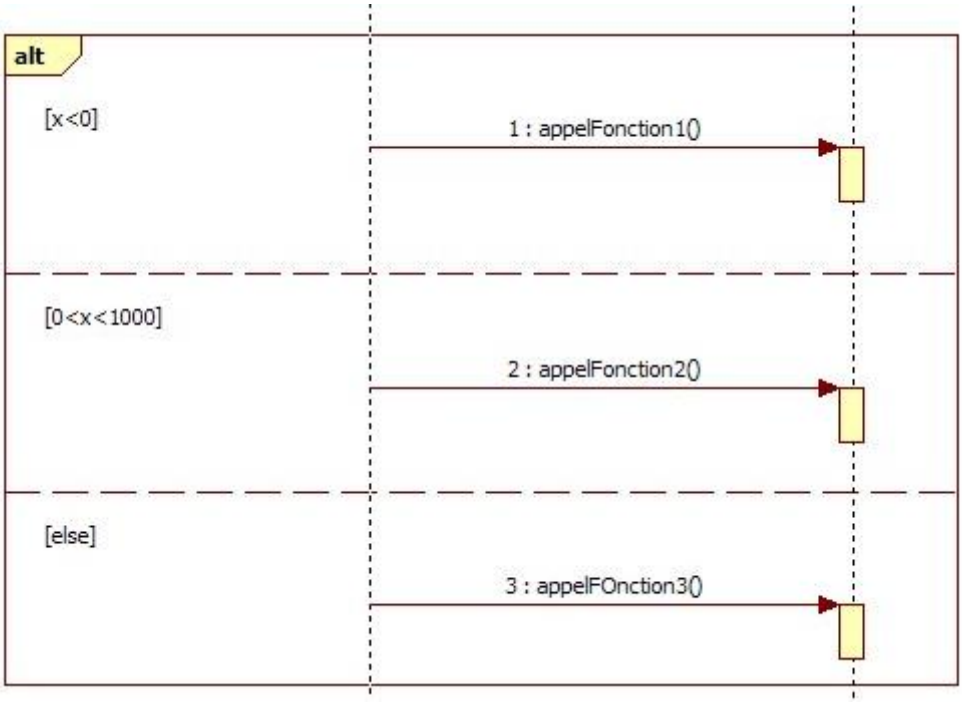


Les 'frame' sont des fragments d'interaction que l'on regroupe pour différentes raisons : soit une exécution conditionnelle, soit une exécution itérée, soit une réutilisation de la frame à un autre endroit. De plus on peut passer des arguments à une frame. Voici la liste des différents opérateurs de frame :

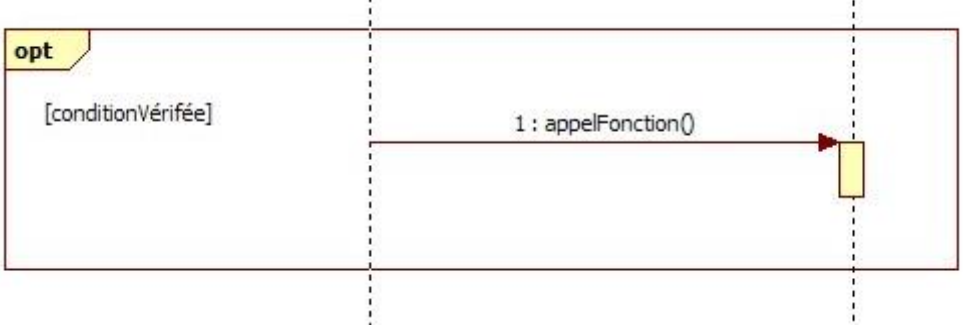
- **sd**: 'sequence diagram', utilisé pour englober une séquence entière d'un diagramme et lui donner un nom (réutilisation)



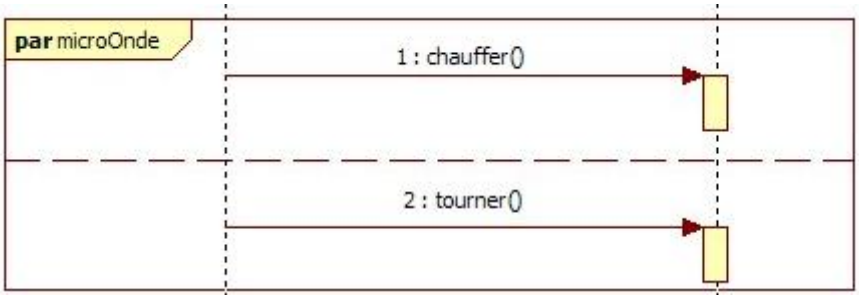
- **alt**: 'alternative multiple fragments', seul le fragment qui vérifie la condition sera execute.



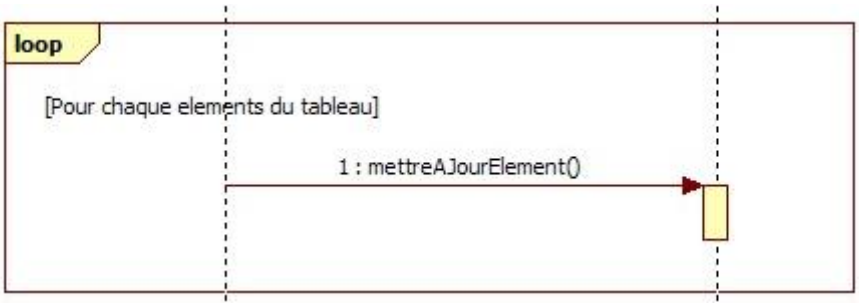
- **opt**: 'optional', si la condition est vérifiée, le fragment sera exécuté (équivalent à un alt avec une seule alternative)

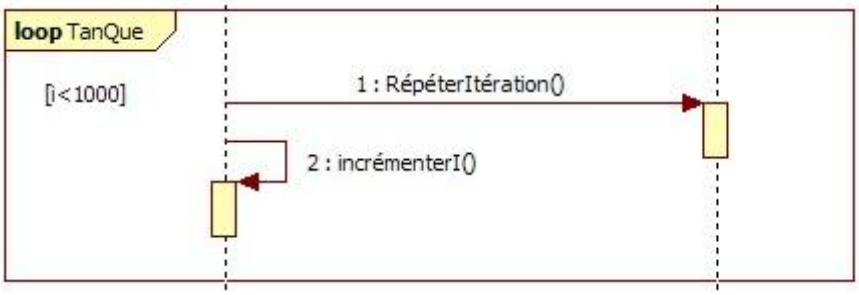


- **par**: ‘parallel’, chaque fragment sera execute en parallèle



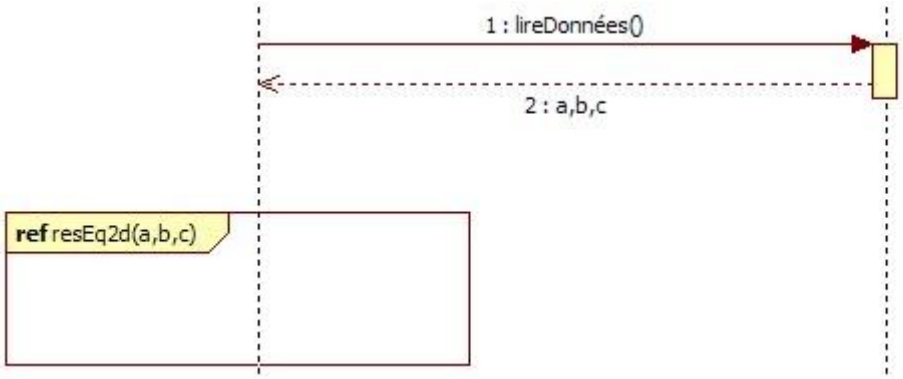
- **loop**: le fragment peut être exécuté plusieurs fois, il est contrôle par un ‘guard’





- **neg**: ‘negative’, le fragment décrit une interaction impossible

- **ref**: ‘reference’, réfère à une interaction définie dans un autre diagramme



- **break:** le fragment est exécuté puis la séquence englobant est terminée

- **critical**: décrit une séquence qui doit être exécutée comme une seule opération, i.e. de manière atomique

VI. *Les diagrammes d'états*

Le but d'un diagramme d'état est de décrire le comportement dynamique d'une classe ou d'un use case. Cela revient à décrire les différents états dans lesquels il peut se trouver, puis à définir les transitions, les gardes, etc …
Pour décrire un état, il faut préciser :

- ✓ Le nom de l'état
- ✓ Activités internes (optionnel)
- ✓ Transitions internes (optionnel)

Une activité interne est une fonctionnalité exécutée par le système ; il en existe de trois sortes :

- ✓ Entrée : Déclenché quand on entre dans l'état
- ✓ Sortie : Exécuté pendant toute la durée ou l'état est actif
- ✓ Pendant : Déclenché quand on sort de l'état

Une transition est une relation entre deux états ; ça représente un changement d'état à partir d'une source vers une cible. Une transition est instantanée et ne peut pas être interrompue.
Pour décrire une transition, il faut préciser :

- ✓ L'élément déclencheur
- ✓ Une condition qui permet la transition
- ✓ Un comportement (exécution d'une activité)

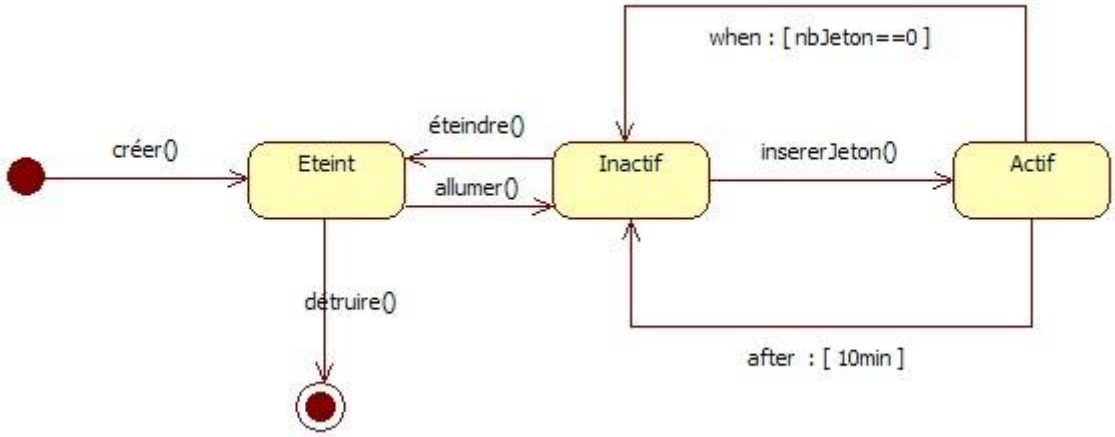
L'élément déclencheur peut être de différents types :

1. Appel d'une opération
2. Signal
3. Condition
4. Un évènement temporel (durée)

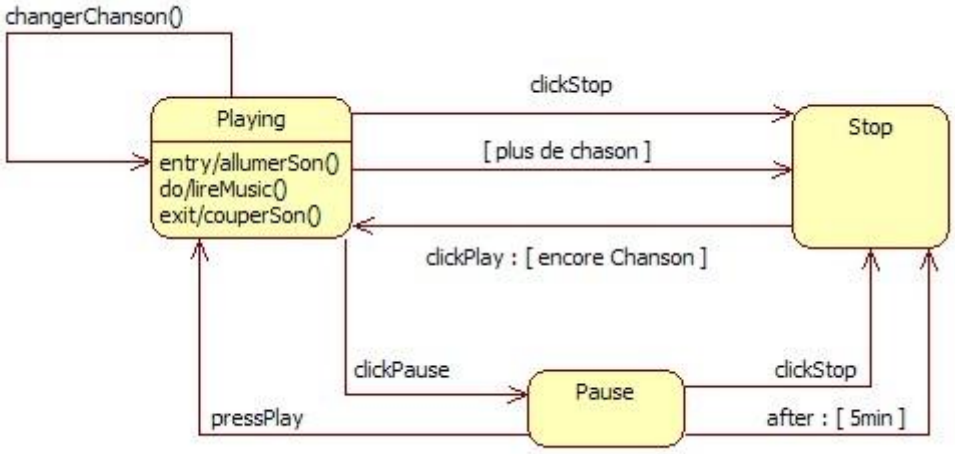
Si aucun évènement n'est spécifié, la transition est exécutée immédiatement après que le comportement interne de la source soit terminé.

Voici quelques exemples :

1. On considère une machine du jeu du palet



2. On considère un lecteur CD



VII. Les interfaces