UML (Non corrigé)

Objectifs

Etude d'un langage objet
Apprentissage d'une méthode d'analyse-conception
modélisation objet

Section

DUT Informatique deuxième année
IUT Caen Campus 3

Bibliographie

Modélisation Objet avec UML par Pierre Muller; Ed Eyrolles

Auteur

Eric Porcq

1) Introduction

1-1) ... Orientée objet

Dans les années 90 , une cinquantaine de méthodes objet ont vu le jour. L'examen de ces méthodes a permis de dégager un consensus autour d'idées communes. Celles-ci s'articulent autour de classes, d'associations (**James Rumbaugh**), de partition en sous-système (**Grady Booch)** et de l'expression des besoins entre l'utilisateur et le système (Les "Use Case" **de I.Jacobson**).

La version 1.0 d'UML a vu le jour en 1997.

2 Les diagrammes d'UML

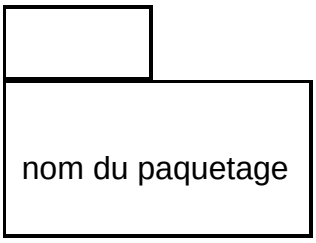
Pour visualiser tous les éléments de modélisation, il existe plusieurs types de diagrammes UML.

On distingue :

- les diagrammes de cas d'utilisation qui représentent les fonctions du système du point de vue de l'utilisateur,
- les diagrammes d'activités qui représentent le comportement d'une opération en terme d'actions,
- les diagrammes d'états-transitions qui représentent le comportement d'une classe en terme d'états,
- les diagrammes de classes qui représentent la structure statique en termes de classes et de relations,
- les diagrammes d'objets qui représentent les objets et leurs relations et correspondent à des diagrammes de collaboration simplifiés, sans représentation des envois de message,
- les diagrammes de collaboration qui sont une représentation spatiale des objets, des liens, des interactions,
- les diagrammes de séquence qui sont une représentation temporelle des objets et de leurs interactions,
- les diagrammes de composants qui représentent les composants physiques d'une application,
- les diagrammes de déploiement qui représentent le déploiement des composants sur les dispositifs matériels,

3) Les paquetages

Les paquetages offrent un mécanisme général pour la partition des modèles et le regroupement des éléments de modélisation. Chaque paquetage est représenté par un dossier.

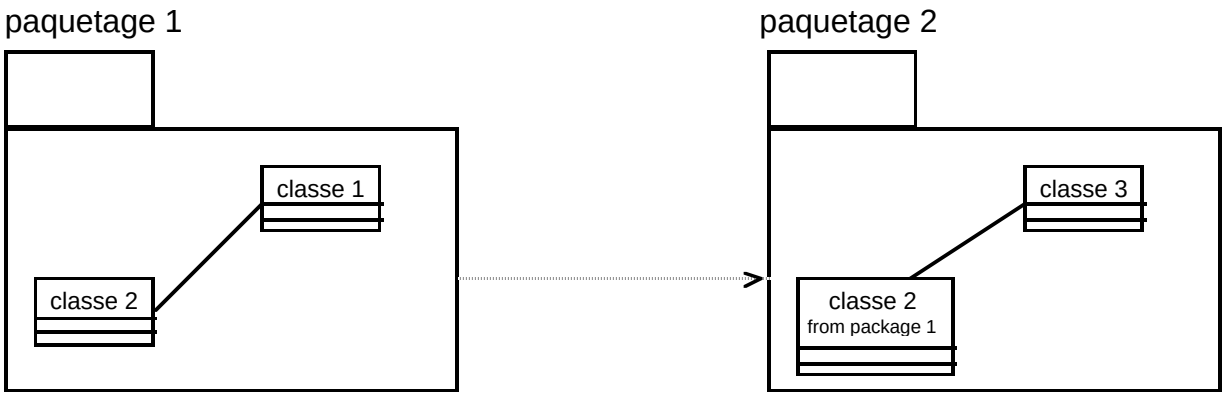


Chaque paquetage correspond à un sous ensemble du modèle et contient classes, objets, relations dont on peut faire des diagrammes UML.

Tout comme avec les systèmes de fichiers, on peut trouver dans un même diagramme des paquetages et d'autres éléments de modélisation.

Une classe contenue dans un paquetage peut se retrouver dans un autre paquetage avec la mention "**importé du paquetage x**" (ou from paquetage x)

Une relation de dépendance entre paquetage signifie qu'au moins une classe d'un paquetage est en relation avec au moins une classe de l'autre paquetage.



4) Les cas d'utilisation

Les cas d'utilisation (use cases) ont été formalisés par Ivar Jacobson. Ils viennent combler un manque dans OMT-1 et Booch 91.

Ils décrivent le système du point de vue de l'utilisateur. Ils permettent de définir les limites du système et les relations entre le système et l'environnement.

Les cas d'utilisation recentrent les besoins sur les utilisateurs. Pour ses raisons, ces diagrammes doivent être formalisés autour du langage naturel; cela permettra aux clients et aux informaticiens de communiquer plus facilement.

Un cas d'utilisation est une manière spécifique d'utiliser un système.

Les diagrammes de cas d'utilisations montrent les interactions entre les acteurs et les cas d'utilisation du système. Ils interviennent tout au long du cycle de développement du système.

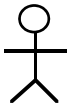
Il doit être lisible par tous (sans formation importante); aussi bien aux analystes qu'aux développeurs et qu'aux clients.

Ils tentent d'éviter aux développeurs des dérives par rapport à l'expression des besoins.

Ils tentent d'éviter aux clients de faire évoluer sans cesse leur besoin

4-1) Les acteurs

- ils sont les utilisateurs du système (clients, pupitreurs, opérateurs, imprimante, routeur...),
- ils ne font pas partie du système,
- ils interagissent avec le système
- le même objet (S.Lebel, La Lexmark Lisieux) peut jouer plusieurs rôles,
- plusieurs objets peuvent jouer le même rôle (étudiant, imprimante),
- ils sont symbolisés graphiquement par petit bonhomme.
- Il peut être lui-même un autre système qui pourrait se décomposer en use-case



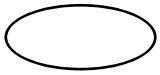
Une fois identifiés, les acteurs doivent être décrits d'une manière claire et concise. Si les acteurs sont nombreux, il est judicieux de les regrouper en catégories.

On peut considérer qu'il existe 4 catégories d'acteurs :

- les acteurs principaux. Ils utilisent les fonctions principales du système
- les acteurs secondaires. Ils utilisent les fonctions secondaires du système (tâches administratives, maintenance, options ...)
- le matériel externe. Il utilise le système et est un acteur incontournable sans quoi, le système ne servirait à rien. Attention, le système informatique sur lequel s'exécute l'application ne peut être un acteur du système : il est le système
- les autres systèmes : ils interagissent avec le système.

4-2) Les cas d'utilisation

- ils regroupent une famille de scénarios
- ils interagissent avec les acteurs ou d'autres cas d'utilisations (utilise ou étendre),
- ils sont symbolisés graphiquement par des ellipses,



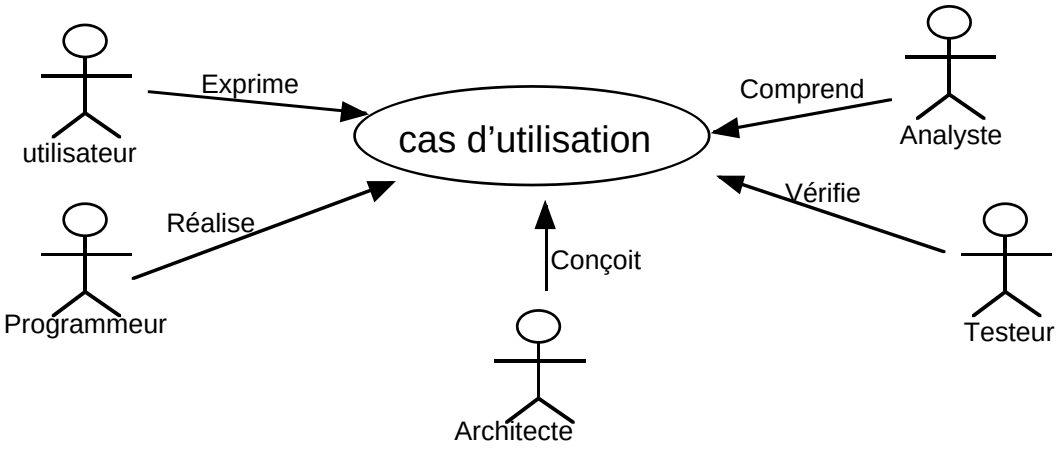
cas d'utilisation

Il faut voir les cas d'utilisations comme des classes dont les instances sont les scénarios.

Les cas d'utilisation se décrivent en termes d'informations échangées et d'étapes dans la manière d'utiliser le système.

Un cas d'utilisation regroupe une famille de scénari d'utilisation selon un critère fonctionnel.

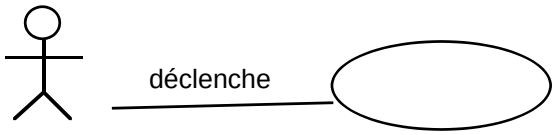
Les scénarios ne sont pas détaillés dans ces diagrammes.



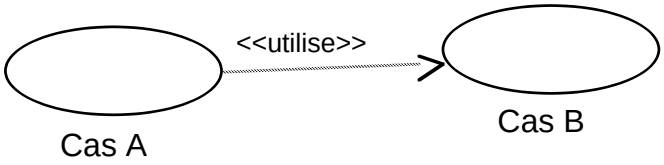
4-3) Les relations

UML définit trois types de relations entre acteurs et cas d'utilisation :

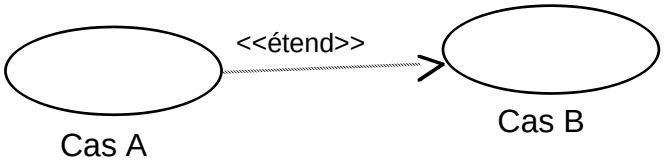
la relation de communication. La participation de l'acteur est signalée par une flèche entre l'acteur et le cas d'utilisation. Le sens de la flèche indique l'initiateur de l'interaction.



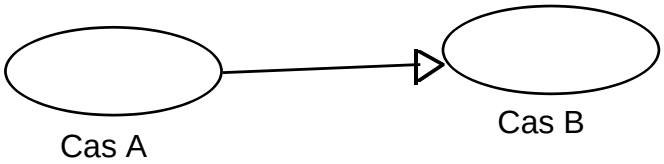
la relation d'utilisation. Elle signifie qu'une instance du cas d'utilisation source nécessite également le comportement décrit par le cas d'utilisation destination.



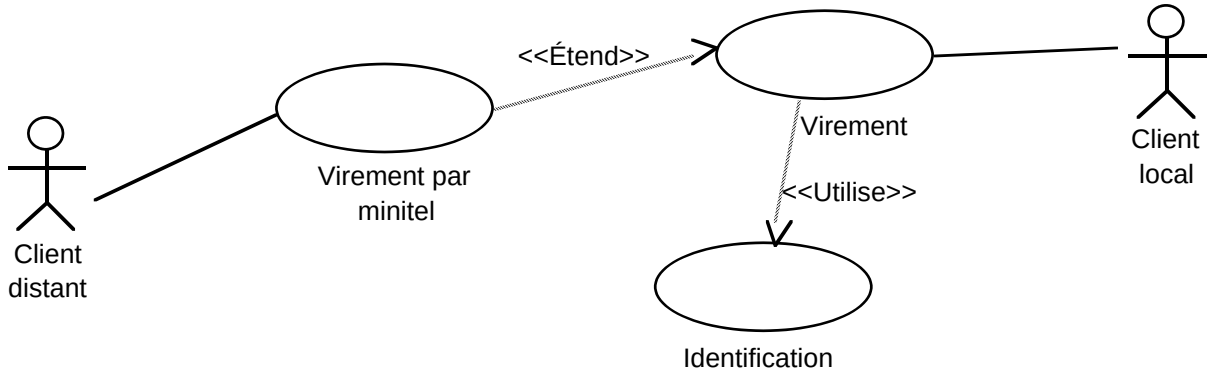
la relation d'extension. Elle signifie qu'une instance du cas d'utilisation source étend (enrichit) également le comportement décrit par le cas d'utilisation destination (on l'emploie aussi pour enclencher des cas optionnels).



la relation de généralisation. Elle permet de regrouper des cas particulier. Attention, ces cas particuliers restent des cas et non des scénarios. Si le cas général est un cas concret, il est préférable d'utiliser la relation d'extension



Exemple : Le virement classique n'est pas abstrait puisqu'il est possible de le réaliser. La relation de généralisation ne s'impose donc pas ici.



4-4) Les limites

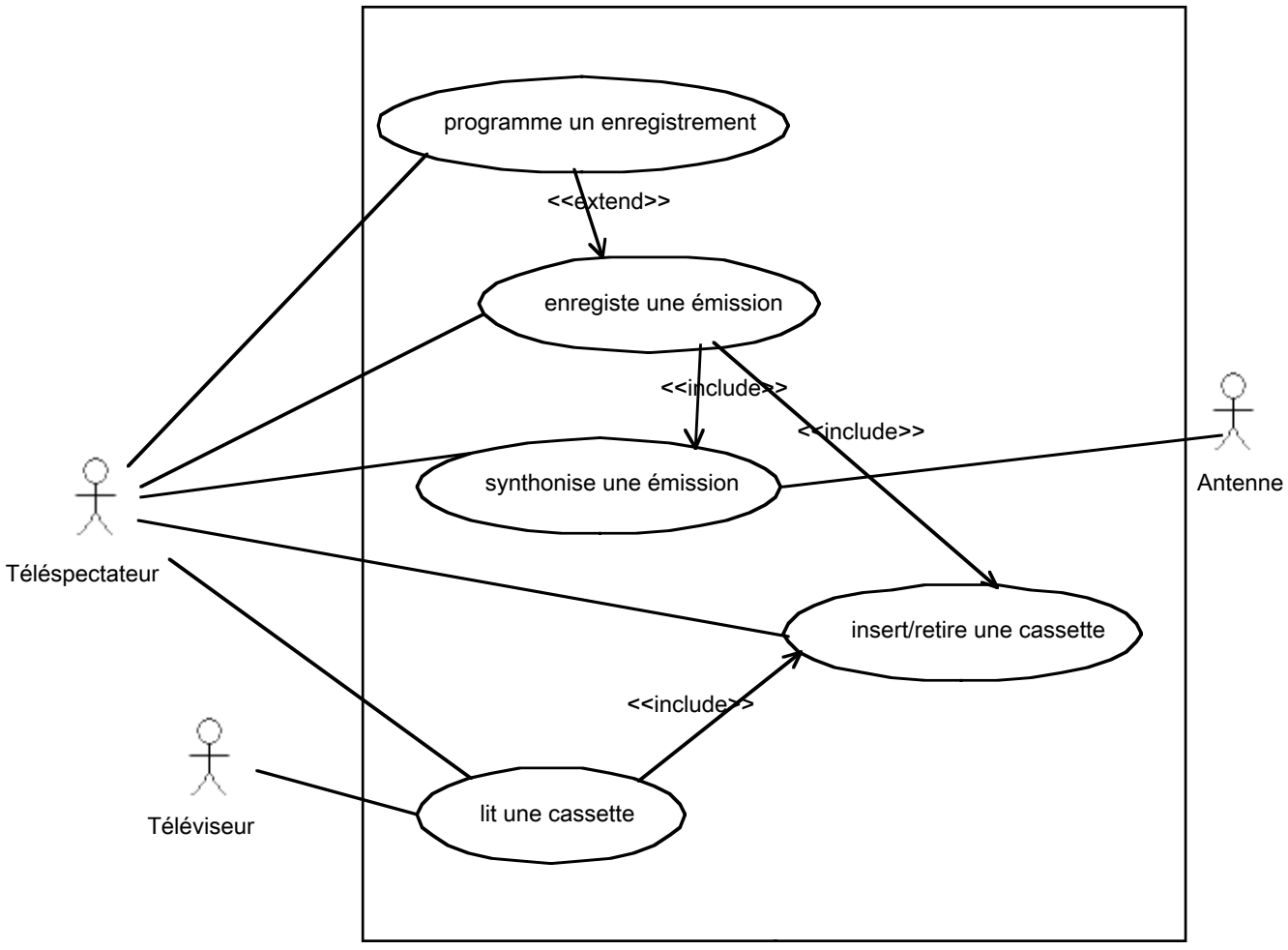
Les diagrammes de cas d'utilisation ne doivent pas devenir des diagrammes d'états-transition, ni des diagrammes de séquence (qui sont des scénarios), encore moins des algorithmes.

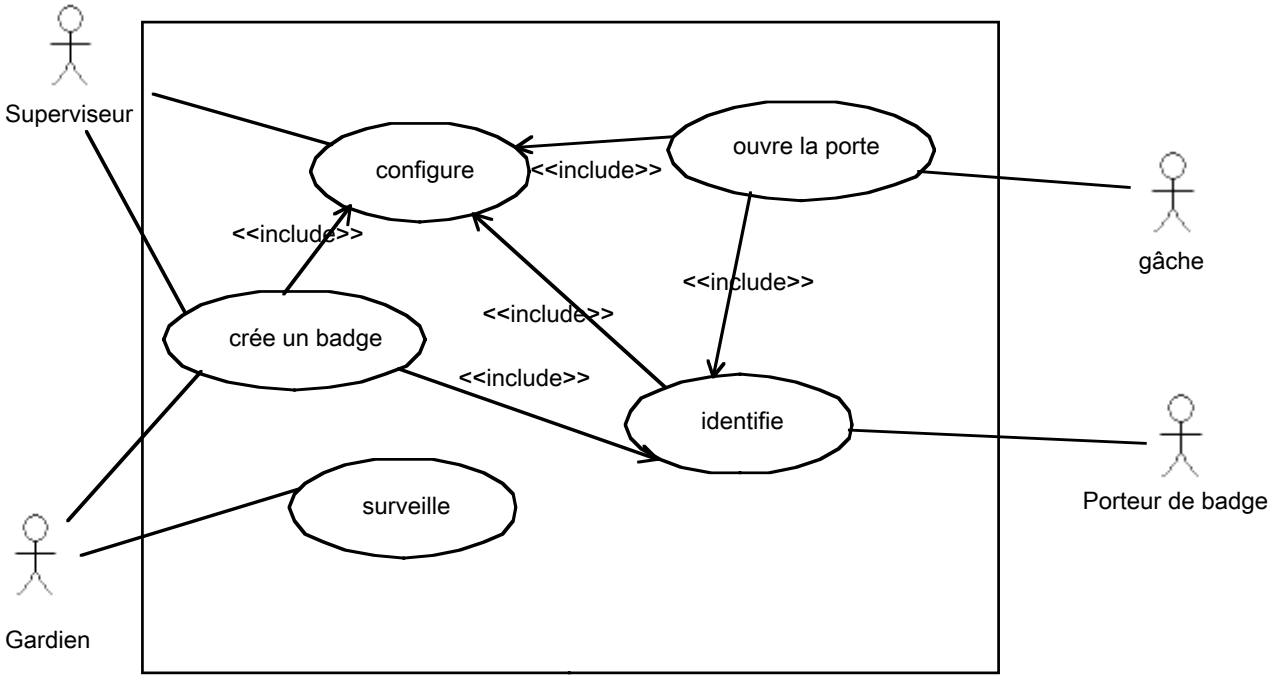
4-5) Transition vers les objets

Les diagrammes de cas d'utilisation répondent à la question quoi ? Mais pas Comment ?

Le passage à l'approche objet s'effectue en associant une collaboration à chaque cas d'utilisation. Une collaboration décrit des objets du domaine, les connexions entre ces objets et les messages échangés. Chaque scénario, instance du cas d'utilisation se représente par une interaction entre les objets décrits dans le contexte de la collaboration (diagrammes de séquence ou de collaboration).

4-6) Exemples





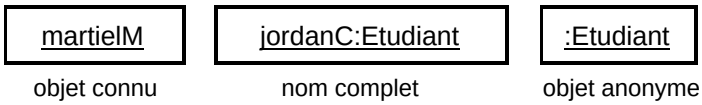
5) Les diagrammes d'objets

5-1) Rappels

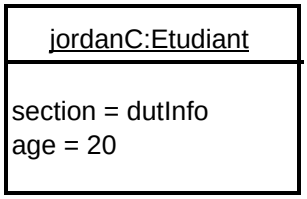
Ils représentent comme pour les diagrammes de classes, la structure statique du système.

Les objets sont reliés par des liens, instances des relations entre les classes des objets considérés.

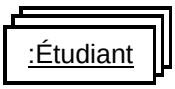
Les représentations graphiques admises sont les suivantes :



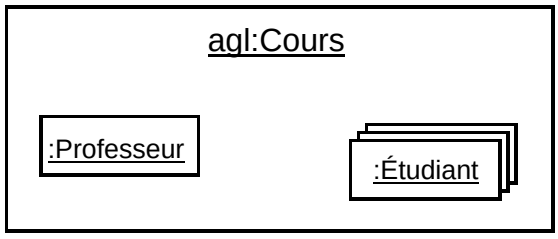
objet connu | nom complet | objet anonyme



instance dont certains attributs sont spécifiés



multiples instances

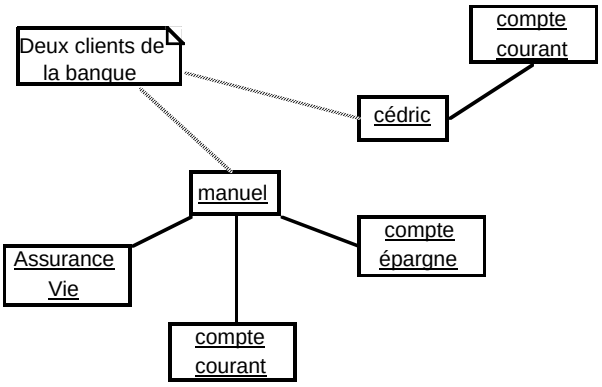


objet composé de plusieurs autres objets : agrégation forte

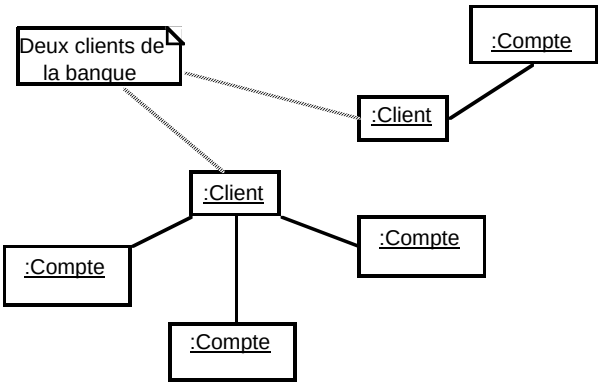
Les objets peuvent être sans *masse* comme les comptes en banque. Dans tous les cas, se sont des représentations abstraites d'un monde réel ou virtuel. Par anthropomorphisme on considère en général que les objets ont une vie propre .

5-2) Diagrammes d'objets

Les diagrammes UML sont en général composés de plusieurs objets possédant des liens entre eux. Des **notes** (rectangles avec un coin replié) peuvent être ajoutés pour améliorer le lisibilité du diagramme.



Il est parfois difficile de trouver un nom à un objet; la notation permet donc d'attribuer un nom **générique** à la place du nom individuel.



le caractère ':' indique qu'il s'agit d'objets anonymes

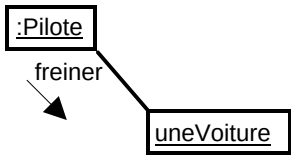
Une classe est une représentation abstraite d'un objet. Elle donne le prototype (type) d'une entité complexe. Elle n'a donc pas d'existence réelle. Pour utiliser un élément typé (comme Individu) dans un programme, on crée une instance d'une classe qui est une variable complexe, image d'une classe, mais ayant une réalité au niveau du programme informatique. L'objet ainsi créé va prendre de la place dans la mémoire de l'ordinateur et, par son comportement , va modifier ses données membres (attributs) ou communiquer avec d'autres objets.

En java, on instancie un objet en exécutant :

```
NomDeLaClasse NomDeLobjet ;
NomDeLobjet = new NomDeLaClasse() ;
```

Son état : il regroupe les valeurs instantanées de tous les attributs ou propriétés d'un objet. Ces derniers sont des informations qui qualifient l'objet qui les contient. Chaque attribut peut prendre une valeur dans un domaine de définition donné.
Rq : il arrive parfois que l'on considère un objet (informatique) comme suffisamment primitif pour ne pas avoir d'état.

Son comportement : il regroupe toutes les compétences d'un objet et décrit les actions et les réactions de cet objet. Chaque atome de comportement est appelé opération ou méthode. Les opérations sont déclenchées suite à une stimulation externe, représentée sous la forme d'un message envoyé par un autre objet.



En réponse à un message, l'objet destinataire déclenche un comportement. Le message est ici un contrôle (sélecteur)

Rq : il arrive parfois que l'on considère un objet (informatique) comme suffisamment statique pour ne pas avoir de comportement.

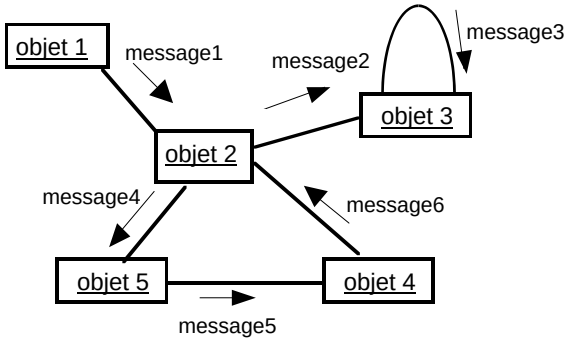
Son identité : elle caractérise son existence propre. L'identité permet de distinguer tout objet de façon non ambiguë même si son état est identique en tout point à celui d'un autre objet. Un objet à toujours une identité.

5-3) Communication entre objets avec UML

Les diagrammes de collaboration montrent des interactions entre objets (instances de classes et acteurs).

Ils permettent de représenter le contexte d'une interaction, car on peut y préciser les états des objets qui interagissent.

Le **message** est le support d'une relation de communication qui relie de façon dynamique les objets séparés par le mécanisme de décomposition. Il se propage naturellement sur un lien entre deux objets.



Les objets communiquent et échangent des messages; des flots de contrôle, des flots de données : datagrammes, appels, événements ...

On peut répertorié cinq catégories de messages :

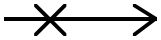
- Les constructeurs,
- Les destructeurs,
- Les sélecteurs (accesseurs en lecture),
- Les modificateurs ou mutateurs (accesseurs en écriture),
- Les itérateurs qui visitent l'état d'un objet complexe (moins employé).

Il existe de même cinq (dont deux principales) méthodes d'envoi de message:

- **simple** : un seul objet est actif. Le symbole est une flèche simple. (UML 1.0)

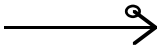
- **asynchrone** : l'émetteur n'est pas bloqué (exécution parallèle des deux objets) (UML 1.0). Le symbole est une demi flèche.

- **synchrone** : L'opération est déclenchée (et l'expéditeur débloqué) que lorsque le destinataire accepte le message. Le symbole est une flèche barrée d'une croix. Le flot de contrôle passe de l'émetteur au récepteur (l'émetteur devient passif et le récepteur actif) à la prise en compte du message.



- **dérobant** : L'opération est déclenchée (et le destinataire débloqué) si le destinataire est en attente de ce message. L'expéditeur n'est jamais bloqué.

- **minuté** : l'expéditeur est bloqué en attente de d'acceptation du message par le destinataire pendant un délai max (timeout).Le symbole est une flèche avec un petit cercle.

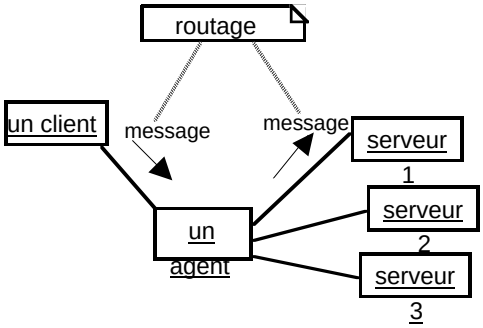


Pour amener à bien la résolution d'un modèle informatique, les objets doivent coopérer entre eux. La modélisation objet consacre donc une grande importance à la communication entre objets. Trois catégories de comportement peuvent être décrit :

Les acteurs sont à l'origine d'une interaction. Ils possèdent un fil d'exécution (thread) et passent la main aux autres objets. Ils sont des objets actifs.

Les serveurs sont destinataires des messages. Ils sont passifs puisqu'ils attendent qu'un autre objet ait besoin de leurs services.

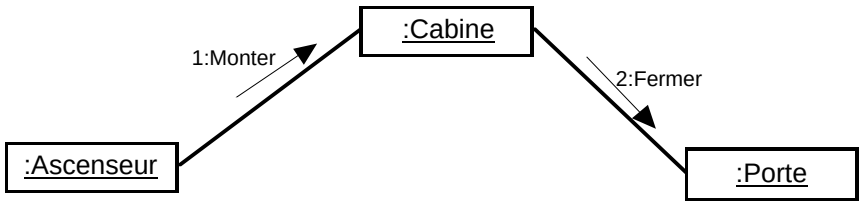
Les agents cumulent les caractéristiques des acteurs et des serveurs. Ils permettent surtout de découpler les acteurs des serveurs ce qui permet à l'acteur d'ignorer le serveur.



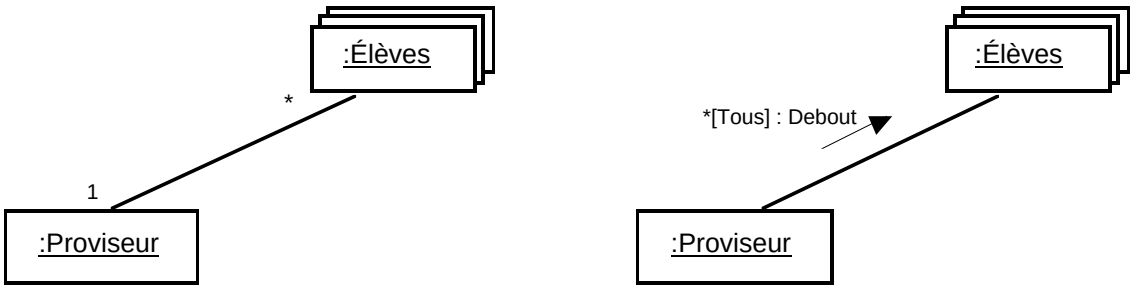
Le client communique indirectement avec le premier serveur, sans le connaître et sans soupçonner l'existence d'autres serveurs.

5-4) Diagrammes de collaboration

Ils insistent plus particulièrement sur la structure spatiale de l'interaction entre les objets.
Ils peuvent être utilisés pour décrire les scénari correspondant aux cas d'utilisation.

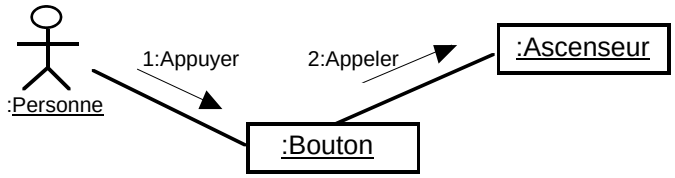


La notation permet de représenter de façon condensée une famille de liens, instances d'une même association.

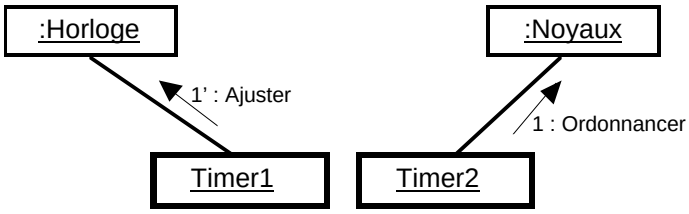


Il est possible de faire figurer un acteur dans le diagramme de collaboration afin de représenter

le déclenchement des interactions par un élément externe au système.

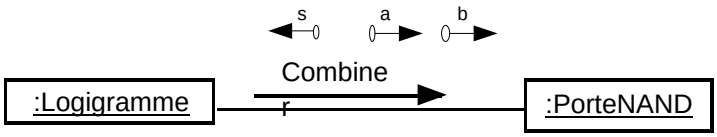


Les objets sont activés par des messages. Dans un environnement multitâche, plusieurs objets peuvent être actifs simultanément. Ils sont représentés par des rectangles plus épais.



Représentation des messages :

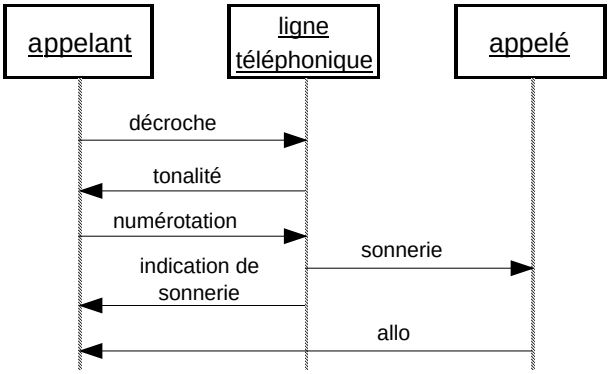
- ils ont un nom. Il s'agit souvent du nom d'une opération de l'objet **destinataire**,
- ils sont représentés par des flèches placées à proximité d'un lien et dirigées vers l'objet destinataire du message. Ils déclenchent une action dans l'objet destinataire,
- ils ont différents modes d'actions (voir chap 5.3)
- les paramètres d'entrée ou de retour sont représentés par des flèches terminées par des cercles.



5-5) Diagrammes de séquence

Ils insistent plus particulièrement sur la structure temporelle de l'interaction entre les objets.
Ils peuvent être utilisés pour décrire les scénari correspondant aux cas d'utilisation.
On distingue deux manières différentes d'utiliser les diagrammes de séquence :

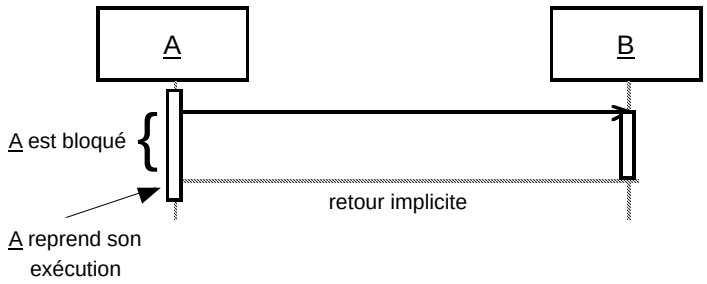
- la documentation des cas d'utilisation. Les termes employés restent proches de l'utilisateur. On n'entre pas dans les détails de la synchronisation. On ne distingue pas les flots de contrôle des flots de données. Les flèches ne sont donc pas encore des messages au sens des langages de programmation.



- la représentation précise des interactions entre les objets. On trouve toutes les formes de communication entre objets : l'appel de méthodes, l'événement discret, le signal, les interruptions matérielles etc...

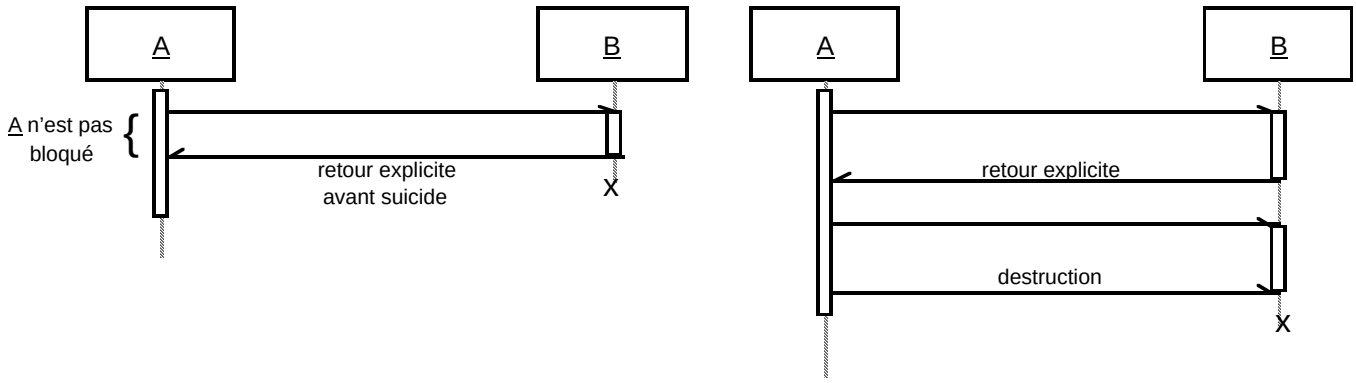
Les périodes d'activité:

Elles sont représentées par des rectangles qui indiquent le début et la fin de l'activité.

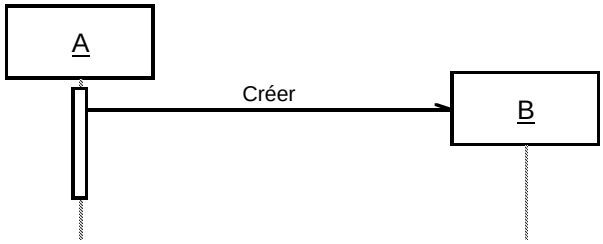


Dans le cas des envois asynchrones, le retour doit être matérialisé lorsqu'il existe. Il faut noter que la fin d'activité d'un objet ne signifie pas la mort de celui-ci.

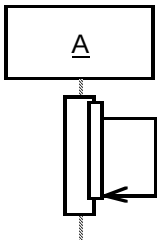
La fin de la vie d'un objet est matérialisée par la fin de la ligne de vie et un X, soit à la hauteur du message qui cause la destruction, soit après le dernier message envoyé par un objet qui se suicide.



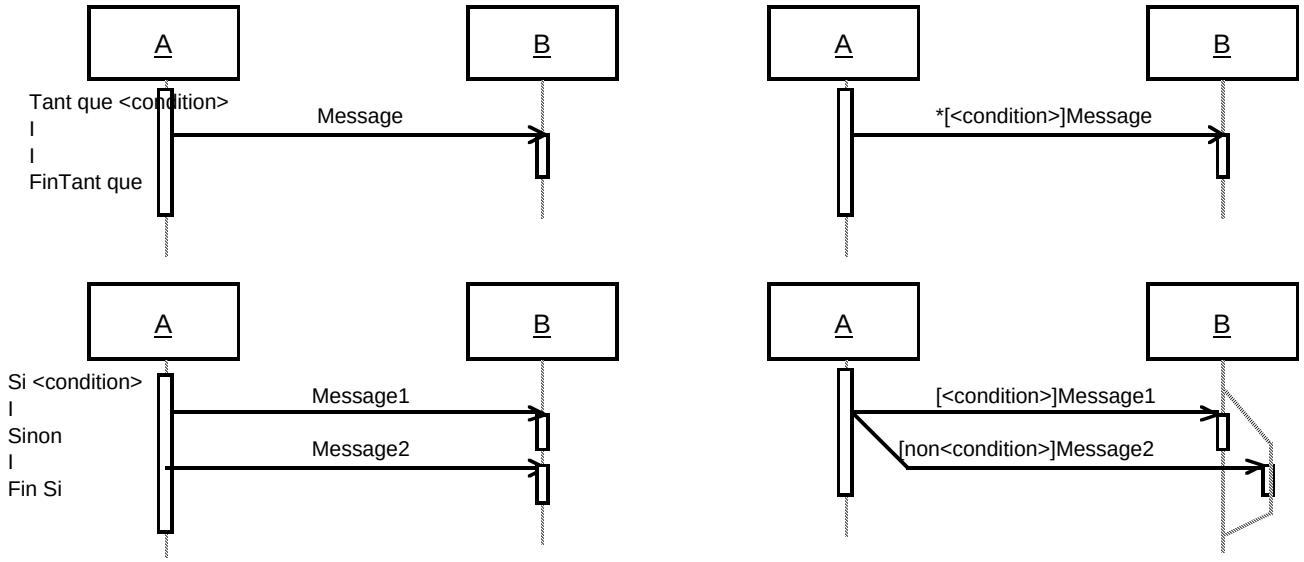
UML 1.0 propose une représentation de la création des objets. (non permise par Rose 98).



Le cas particulier des envois de messages récursifs se représente par un dédoublement de la bande rectangulaire. L'objet apparaît comme s'il était actif plusieurs fois.



L'ajout de pseudo-code sur la partie gauche du diagramme permet la représentation des boucles et des branchements. Cela peut éviter parfois deux diagrammes différents pour deux scénarios proches.

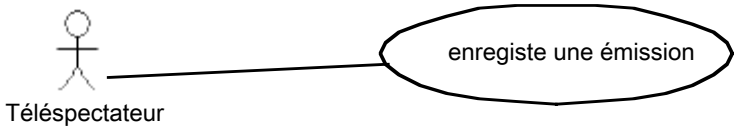


6) Les scénarios

6-1) Définition

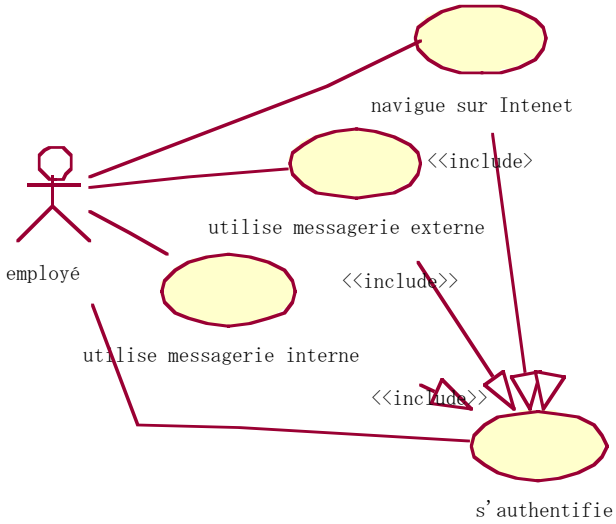
Un scénario est une instance d'un cas d'utilisation. Il décrit un exemple d'interaction possible entre le système et les acteurs. C'est une situation possible pour un cas.

Exemple1 : Etude du cas "Enregistre une émission"



Scénario 1 : Le téléspectateur a placé une cassette de 120 min accessible en écriture et correctement réembobinée; il sélectionne un programme (qui dure 1h30) et lance l'enregistrement...
Scénario 2 : Le téléspectateur a placé une cassette de 120 min accessible en écriture et correctement réembobinée; il sélectionne un programme (qui dure 2h30) et lance l'enregistrement...

Exemple2 : Etude du cas "s'authentifie"



Scénario 1 : l'employé passe sa carte et tape un nom et un mot de passe correct.
Scénario 2 : l'employé n'insère pas (ou mal) sa carte ou celle-ci est illisible
Scénario 3 : l'employé n'indique pas de nom, de mot de passe ou un nom non conforme
...

Principe:

- Définir plusieurs scénarios pour un cas d'utilisation
- un scénario représentant l'exécution nominale;
- des scénarios d'exception (qui ne permettent pas de terminer correctement le cas d'utilisation)

Remarque: Un cas d'utilisation est le regroupement de plusieurs scénarios. Formalisation : Un scénario est f normalisé par un diagramme d'interaction (diagramme de séquence ou diagramme de collaboration).

6-2) Les limites

1 cas d'utilisation peut donner lieu à de nombreux scénarios et donc à de nombreux diagrammes de séquences.

Plusieurs cas => trop de diagrammes ?

Il est possible de regrouper des scénarios proches dans un seul diagramme d'interaction. La norme UML 2.0 préconise (je crois) ce genre de modélisation.

Avantage : génération de code possible à partir d'un seul diagramme et pas à partir de plusieurs. Donc, intéressant en phase de conception détaillée.

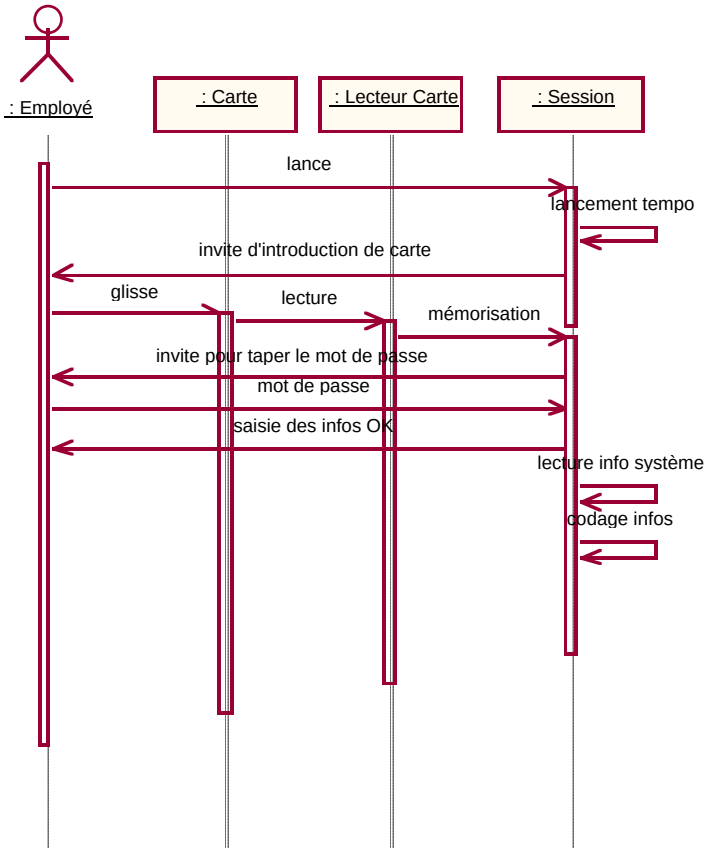
Inconvénient : un seul diagramme complexe nuit à l'analyse et à la compréhension du système

6-3) Les diagrammes de haut niveau

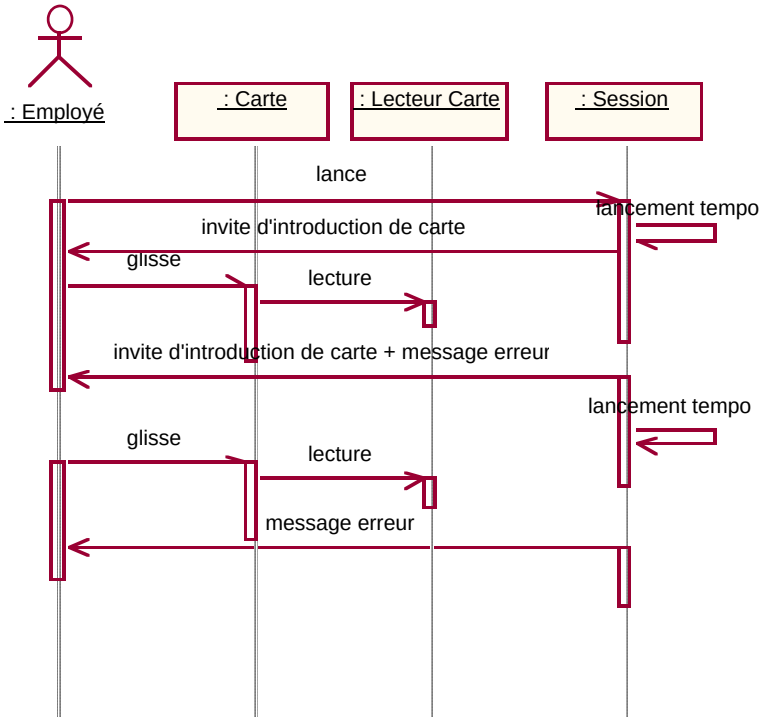
Ils servent à

- mettre en oeuvre les cas d'utilisation
- expliquer le comment pour un cas qui exprime le quoi ?
- documenter le dossier de spécifications

Scénario 1 : l'employé passe sa carte et tape un nom et un mot de passe correct.



Scénario 2 : l'employé n'insère pas (ou mal) sa carte ou celle-ci est illisible



6-4) Méthode de conception employée dans le TD AGL

On a réalisé :

- L'étude générale du système
- le diagramme des cas d'utilisation
- la spécifications des technologies et matériels imposés par le client

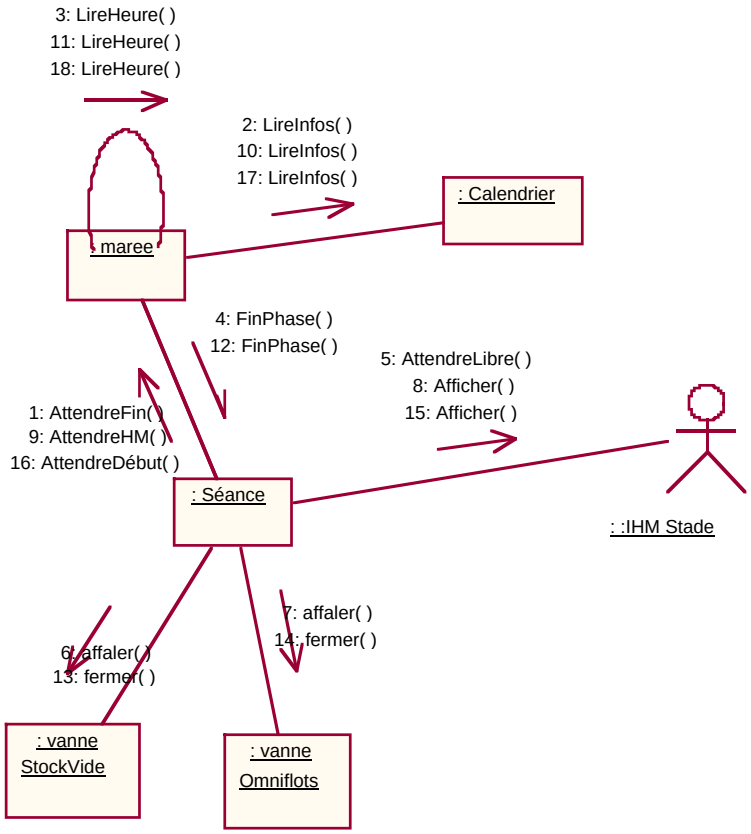
On va :

- rechercher la liste des objets physiques et technologiques présents sur le système
- le tri de ces objets pour découvrir les objets **informatiques** (dont le comportement est géré par le système). un objet informatique pourra avoir la même identité qu'un objet physique (Carte...) même s'ils ne sont que la modélisation du comportement et de l'état de cet objet.
- un premier diagramme d'objet sans interaction (ou un diagramme statique sans relation, sans propriété et sans opération)

6-5) Les diagrammes de bas niveau

Ils servent à :

- mettre en oeuvre les cas d'utilisation
- expliquer le comment pour un cas qui exprime le quoi ?
- déterminer les opérations (méthodes) des classes
- compléter le diagramme statique
- générer le squelette de l'application (si l'AGL le permet)



6-6) Transition vers le diagramme statique

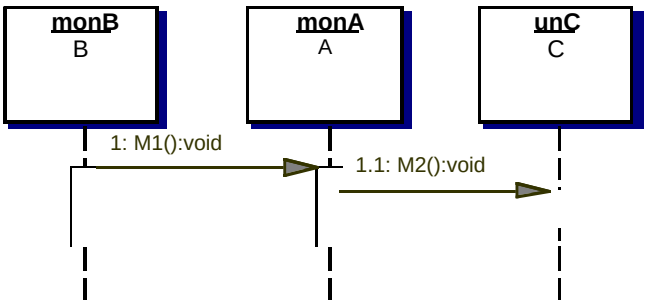
Chaque diagramme d'interaction va venir alimenter le diagramme statique qui va se compléter et s'affiner au fur et à mesure que des scénarios seront composés.

Quand tous les scénarios principaux de tous les cas auront été réalisés, le diagramme statique sera terminé en grande partie.

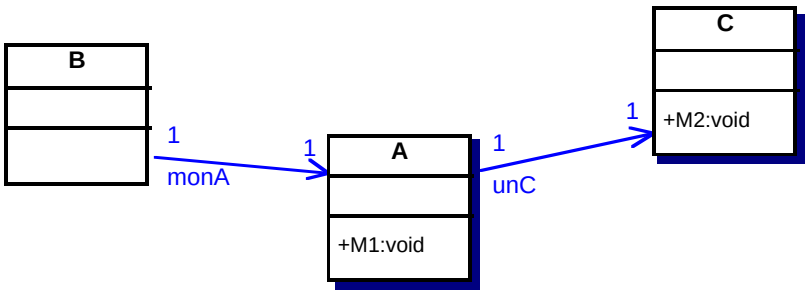
Il restera à étudier "les scénarios catastrophes" pour permettre de réaliser la gestion des exemptions

On pourra aussi faire apparaître des classes "spéciales" (d'associations, d'interfaces, généralisées ...)

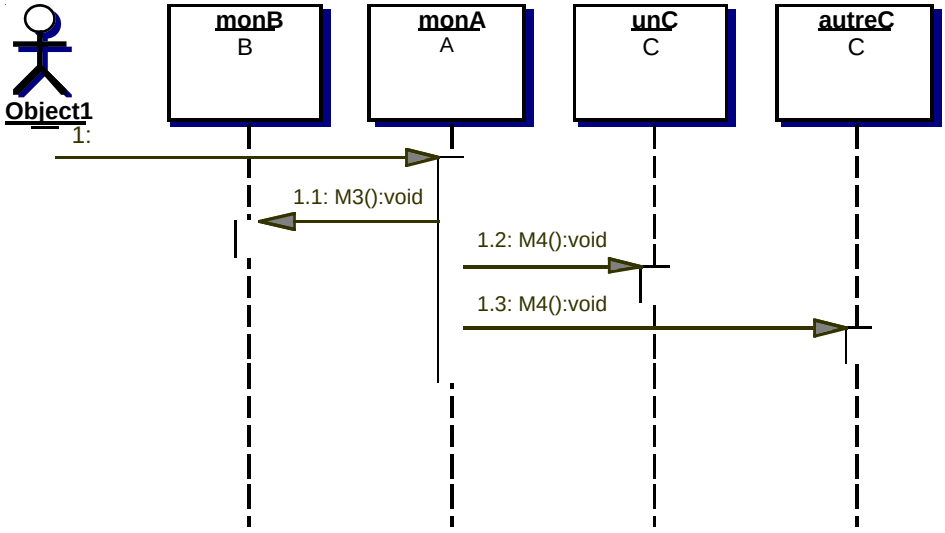
premier scénario :



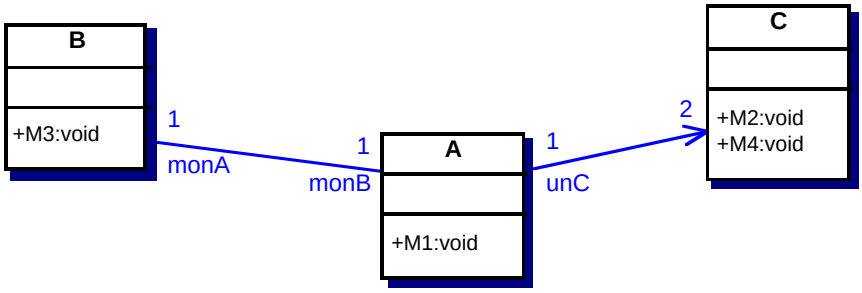
Résultat 1 :



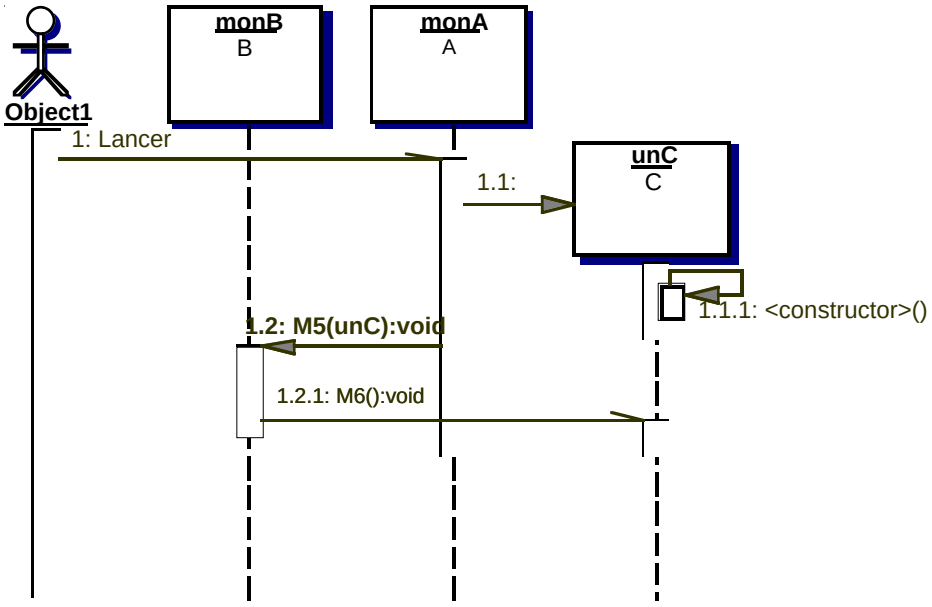
second scénario :



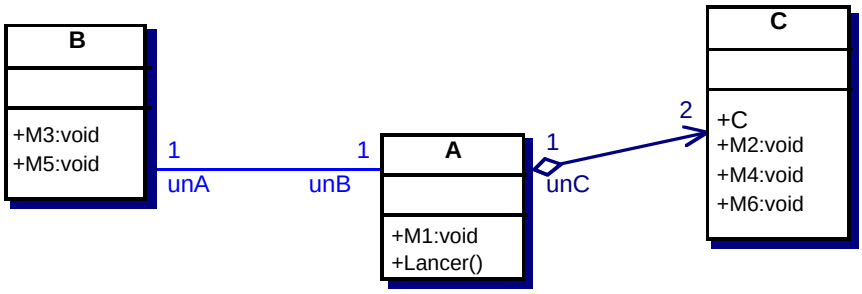
Résultat 2 :



troisième scénario :



Résultat final :



Les scénarios successifs ont permit d'aboutir à un diagramme de classe complet (avec méthodes, navigabilité, cardinalités, agrégations ...)

Exercice 1 : donner le code java correspondant au scénario 3

Code Java :

```
public class Appli
{
  A monA;
  B monB;
  public Appli()
  {
    monB = new B();
    monA = new A(monB);
    monA.Lancer();
  }
  public static void main(String[] args)
  {
    Appli appli = new Appli();
  }
}


class A
{
  C unC;
  B unB;
  public A(B b)
  {
    unB = b;
  }
   public void Lancer()
  {
    unC = new C();
    unB.M5(unC);
  }
}

class B
{
  public void M5(C c)
  {
    c.M6();
  }
}

class C
{
  public C()
  {
  }
  public void M6()
  {
  }
}
```

7) Les diagrammes d'états transitions

Les diagrammes d'états transitions décrivent le comportement dynamique d'une classe. Ils montrent des automates d'états finis.

Avec les scénarios et les diagrammes d'interaction, on a pu compléter le diagrammes statique avec les asscociations et les opérations. Grâce au diagramme état tansition, on va compléter les propriétés des objets et ainsi, "*terminer* " le diagramme statique.

7-1) Les automates

Le comportement des objets d'une classe peut Etre décrit de manière formelle en termes d'**états** et d'événements**,** au moyen d'un **automate** (ou machine) relié à la classe considérée.

Exemple : classe porte

états possibles : fermée, en ouverture, ouverte, en fermeture

Les systèmes purement combinatoires n'ont pas d'automate.

Un automate est une abstraction des comportements possibles, à l'image des diagrammes de classes qui sont des abstractions de la structure statique. Chaque objet suit le comportement associé à son automate.

Les automates peuvent parfois être utilisés pour décrire un groupe d'objets.

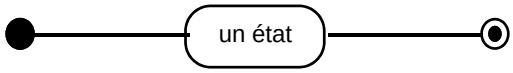
Les automates et les scénarios sont complémentaires.

7-2) Les états

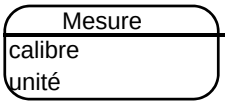
Chaque objet est à un moment donné dans un état particulier. Les états se caractérisent par la notion de durée et de stabilité. **La présence ou non des liens (des références au niveau coneption) avec les autres objets et la valeur des attributs d'un objet déterminent son état.**

Un état est représenté sous la forme d'un rectangle arrondis et d'un nom qui l'identifie.

Pour éviter toute ambiguïté, un état initial et un (ou plusieurs) état(s) final(aux). Il n'y a parfois pas d'état final. L'état initial se représente sous la forme d'un point noir. L'état final se représente sous la forme d'un point noir encerclé.



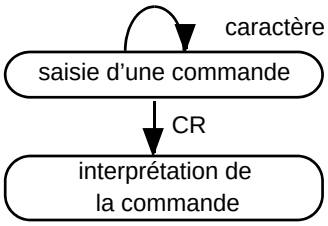
Les états peuvent éventuellement contenir des variables exprimées sous la forme d'attributs. Les variables d'état appartiennent à la classe associée à l'automate, mais peuvent Etre représentées dans les diagrammes d'états-transitions lorsqu'elles sont manipulées par les actions ou les activités.



7-3) Les transitions

Lorsque les conditions dynamiques évoluent, les objets changent d'état en suivant les règles de l'automate associé à leurs classes. Les états sont reliés par des connexions **unidirectionnelles** appelées transitions. On considère que le passage d'un état à un autre est **instantané** de sorte que l'objet est toujours dans un état connu.

Certaines transitions maintiennent l'objet dans le même état : transition réflexive.

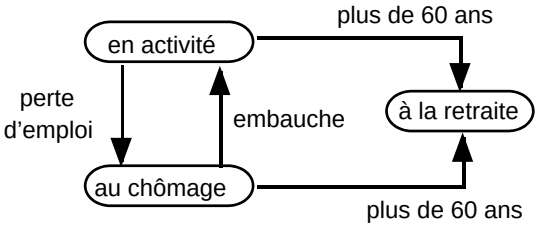


7-4) Les événements

Un événement est l'occurrence d'une situation concernant l'objet. Contrairement à l'état qui dure, l'événement est instantané. Il est le déclencheur d'une transition d'un état vers un autre.

La spécifications de l'événement comporte :

- son nom
- la liste des paramètres
- l'objet expéditeur
- l'objet destinataire
- la description de la signification de l'événement



Il existe dans UML différents types d'événements :

1. **réception d'un signal**. C'est un message asynchrone entre deux instances qui provoque cet événement. Ex : réception d'un code d'erreur dans une boîte à lettres.
2. **appel d'opération** : c'est un message simple ou synchrone entre deux instances qui provoque cet événement. Ex : création d'un objet
3. **modification d'une expression booléenne** liée au changement d'une propriété d'un objet. Le changement d'état est irréversible. On associe le mot clé quand à cette condition booléenne. Ex : quand (cassette = terminée)
4. **temporel** : expiration d'un délai ou occurence d'une date absolue. On associe le mot clé quand ou après à cet événement. Ex : quand (date fin d'enregistrement = "14h20") ou après (1h30 de durée d'enregistrement)

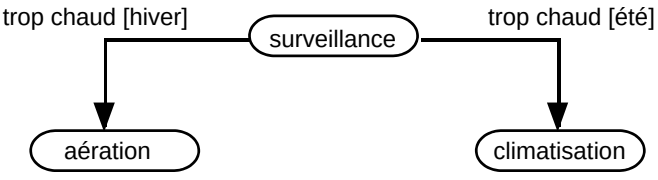
Pour les deux permier types d'événements , la syntaxe est : <nom de l'événement>(<paramètres>)

Ex : Démarrage(vitesse)

Un événement peut évidemment provenir d'une autre classe.

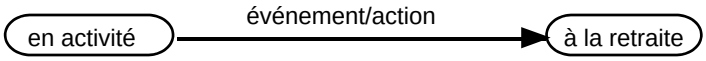
7-5) Les gardes

Une garde est une condition booléenne qui valide ou non le déclenchement d'une transmission lors de l'occurrence d'un événement. Les gardes (pour un même événement) sont exclusives entres elles.



7-6) Les opérations,actions,activités

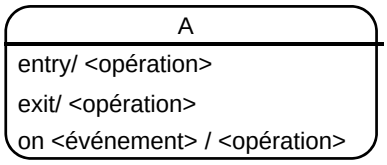
Le lien entre les opérations des diagrammes de classe et les événements des diagrammes d'états sont représentés par les actions et les activités.



L'action correspond à une opération de l'objet destinataire de l'événement. Si l'ation est une opération de la classe dont on étudie les états, elle peut avoir accès aux paramètres de l'événement.

Les états peuvent également contenir des actions; elles sont exécutées :

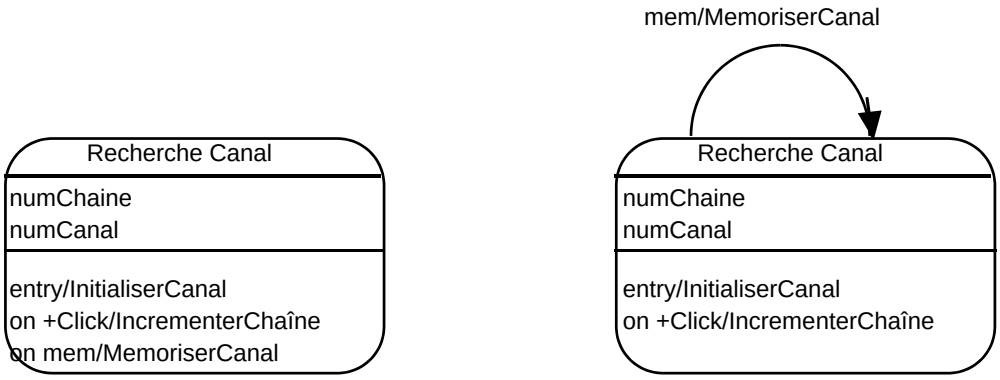
- à l'entrée de l'état : entry:
- à la sortie de l'état : exit:
- à l'occurrence d'un événement de l'état : on <un événement>:



Un événement interne active une action qui n'entraîne pas de changement d'état.

Attention : il y a une différence entre une transition interne et une transition réflexive. La première n'entraîne pas l'exécution des actions d'entrées et de sorties.

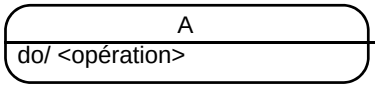
Exemple :



dans le diagramme de droite, l'appui sur le bouton mem va Ré-initialiser le canal (à 0).

Les actions correspondent à des opérations dont le temps d'exécution est négligeable devant l'automate.

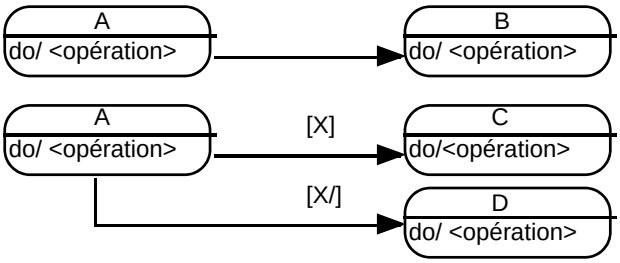
Une opération qui prend du temps correspond plutôt à une activité. Le mot clé **do:** indique une activité. Cette dernière correspond forcément à un état.



Contrairement aux actions, les activités peuvent Etre interrompues à tout moment, dès qu'une transition de sortie de l'état est déclenchée.

Certaines activités sont cycliques et ne s'arrêtent que lorsqu'une transition de sortie est déclenchée. D'autres activités sont séquentielles et démarrent à l'entrée de l'état (ou tout de suite après l'exécution des actions d'entrées).

Lorsqu'une activité séquentielle arrive à son terme, l'état peu Etre quitté par une transition automatique (sans événement déclencheur).

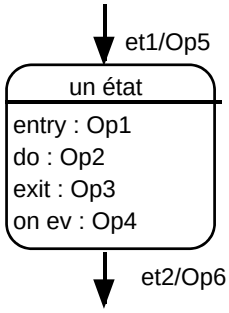


des gardes peuvent éventuellement être placées, même sans événement

Il est possible d'indiquer l'invocation d'un sous-graphe. Le mot cle est **include**

En résumé, une opération peut être associée à une :

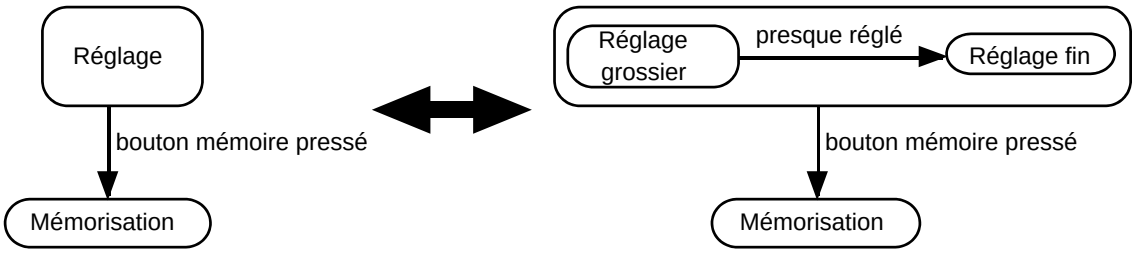
- action de transition d'entrée (Op5)
- action d'entrée (Op1)
- activité dans l'état (Op2)
- action de sortie (Op3)
- action d'un événement interne (Op4)
- action de transition de sortie (Op6)



7-7) Etats composites (ou super-états)

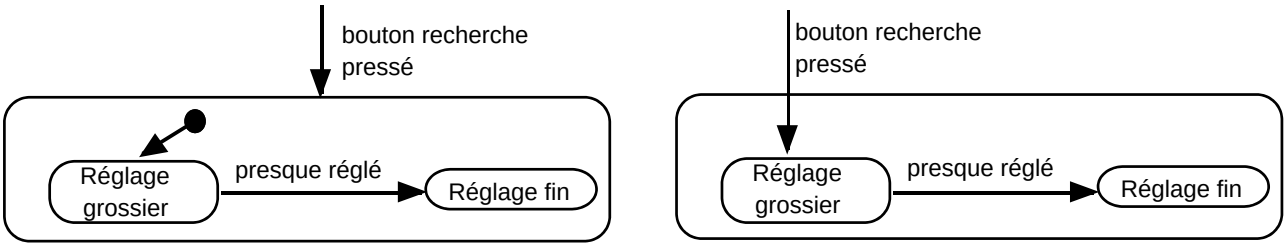
Pour simplifier des diagrammes comportant beaucoup d'états, il est possible de créer des états composites qui peuvent se décomposer en sous-états qui peuvent eux mêmes ...

Un état composite peut se décomposer en plusieurs sous-états. L'objet ne peut être que dans un seul de ses sous-états à la fois.

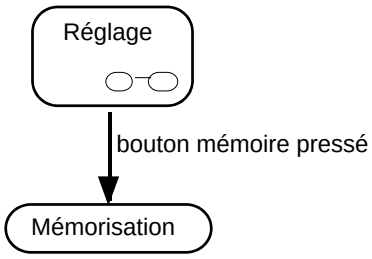


Une transition de sortie d'un état composite s'applique à tous les sous états.

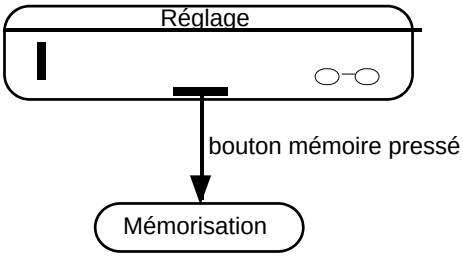
Une transition d'entrée ne s'applique qu'à un seul sous état



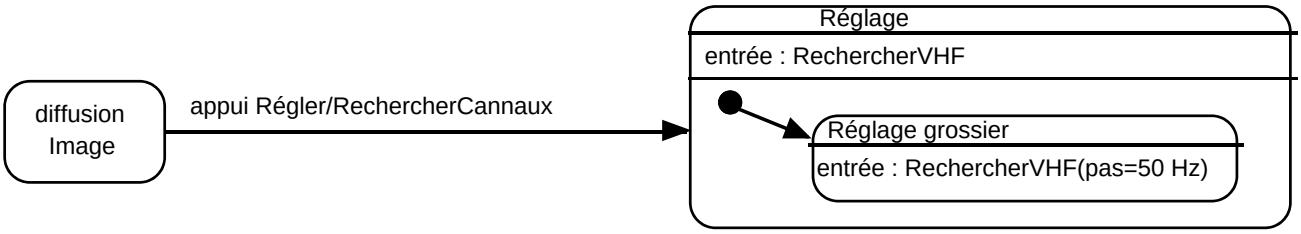
Pour montrer qu'un état est composite, on peut ajouter un petit symbole composé de deux états et d'une transition.



Il est également possible de simplifier la représentation d'un état composite en remplaçant les sous-états par des **souches** On sait toujours qu'il y a des sous-états (et leur nombre) sans connaitre ces sous-états. *Ce n'est pas la simplification la plus pertinente.*



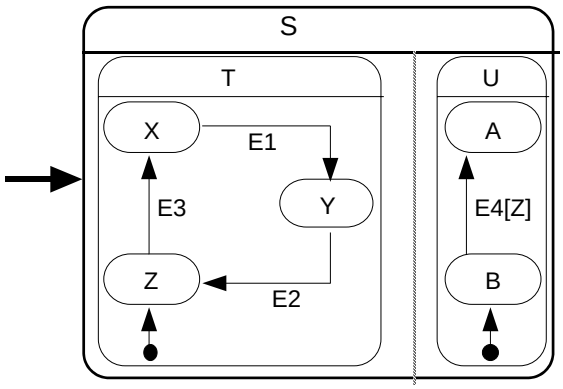
Les actions d'entrées respectent la hiearchie de la décomposition.



dans l'exemple ci-dessus, les actions s'exécuteront dans l'ordre : RechercherCannaux, RechercherVHF, RechercherVHF(pas=50Hz)

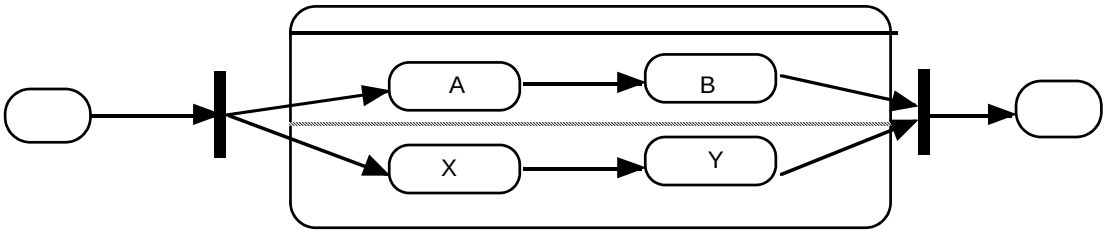
7-8) Etats concurrents

Un état peut être composé de plusieurs sous-états. A la différence avec les états composites, l'objet dans ce cas est dans tous les sous états à la fois . On place un pointillé pour séparer les diverses régions de l'état.



La transition entrante dans l'état S implique l'activation simultanée des automates T et U, c'est à dire, dans l'état composite (Z,B). Les automates peuvent évoluer indépendamment ensuite.
On sortira de l'état S à la sortie du dernier automate T ou U ou par un événement.
On peut ajouter une garde pour indiquer une dépendance entre les régions

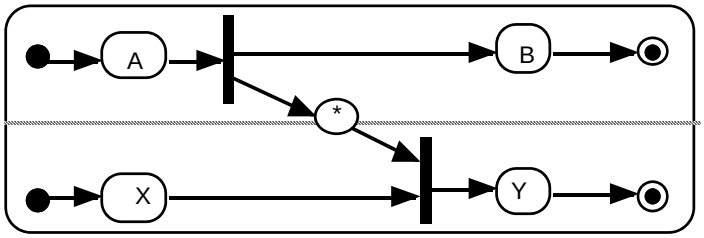
Une transition concurrente peur avoir plusieurs états source et destination. Des barres de synchronisation permettent les jointures et les synchronisations.



- La barre de synchronisation permet de représenter graphiquement des points de synchronisation.
- Les transitions automatiques qui partent d'une barre de synchronisation ont lieu en même temps.
- On ne franchit une barre de synchronisation qu'après réalisation de toutes les transitions qui s'y rattachent.

7-9) Etats de synchronisation

Ils sont utilisés pour synchroniser des états concurrents. Des cercles indiquent ces états de synchronisation. Une valeur peut limiter le nombre de déclenchement de l'état (*=infini).

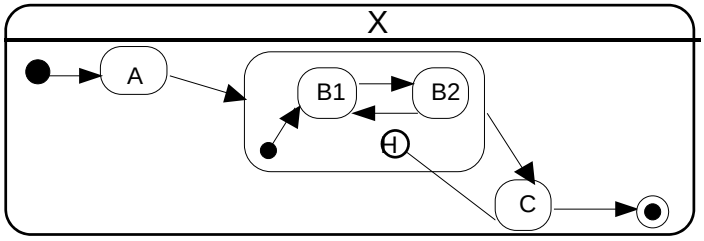


Dans cet exemple, A et X sont concurrents. Mais Y et A ne peuvent pas l'être puisque l'activtion de Y est issue de la transiton de sortie de A et X. B peut être concurrent à X.

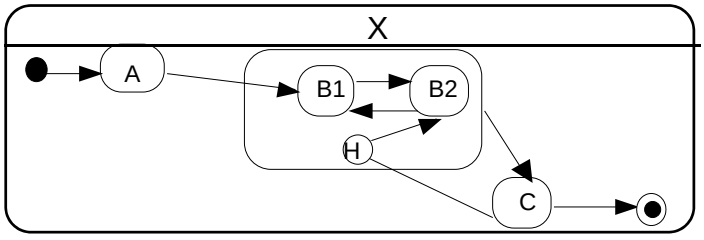
7-10) Etats historiques

Il est possible de mémoriser le dernier sous état activé dans un état composite. Une nouvelle activation de l'état activera à nouveau ce sous état (avec exécution de toutes les actions d'entrées).

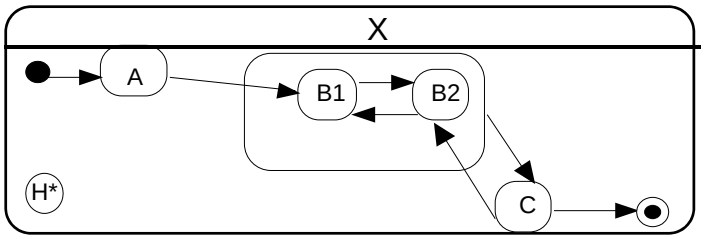
Un **H** entouré d'un cercle symbolise qu'un état composite garde un historique de l'activation. On peut placer un H à n'importe quel niveau d'un état composite.



Si A active B, c'est d'abord B1 qui sera activé. Puis, lors des activations suivantes, l'activation de B activera le dernier sous cas activé (B1 ou B2). Ici, seul le cas B possède un historique.



En ajoutant une transition de sortie sur l'historique, on peut indiquer le sous activé par défaut.



Le symbole H* indique que tous les sous-états gardent un historique des activations.

7-11) Exercices

1)Etude d'une porte de garage télécommandée :

Un local est muni d'une porte d'accès. Cette dernière peut être manipulée manuellement, et être ainsi placée dans toute position, y compris les positions extrêmes "haute" et "basse", la porte sera considérée comme ouverte si elle se trouve en position haute et fermée lorsqu'elle se trouve en position basse.

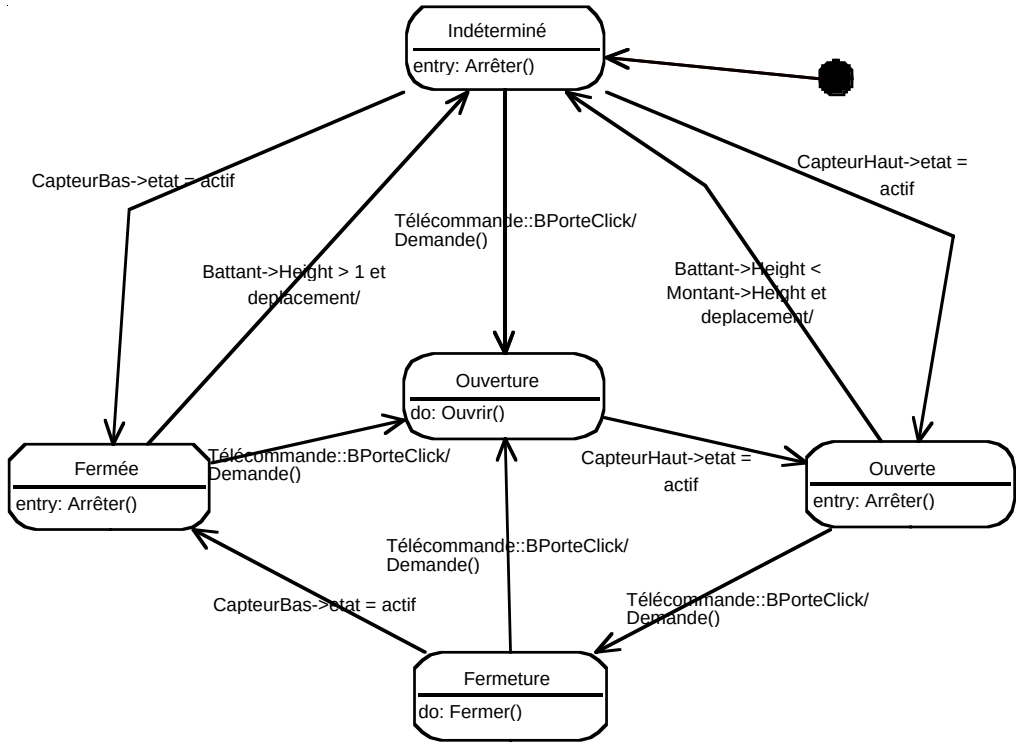
Un opérateur est muni d'un boîtier de télécommande comportant un bouton de commande, ce dernier lui permet d'actionner la porte à distance.

La porte reçoit l'événement de commande ; selon sa position ("fermée" ou "ouverte") la porte s'ouvre ou se ferme (jusqu'à sa position finale).

Si l'événement arrive lors de l'ouverture de la porte, il est sans effet.

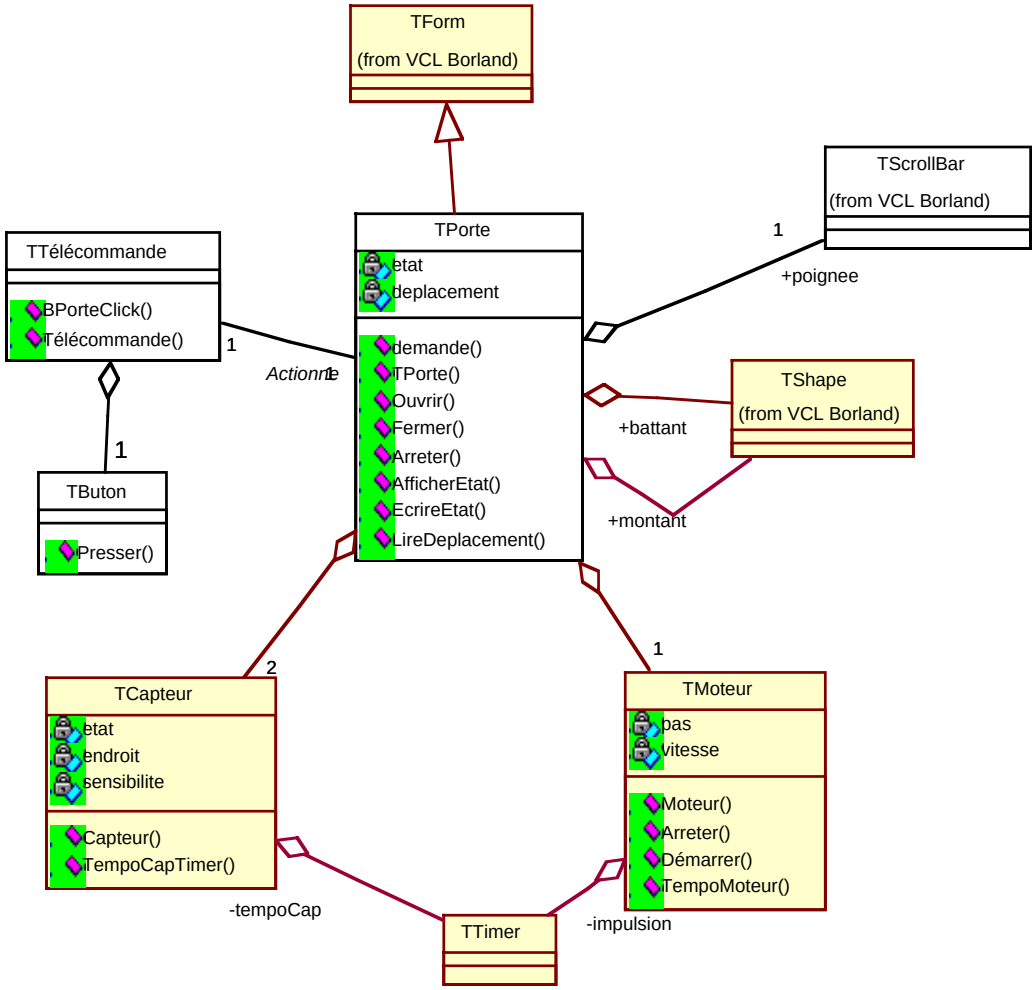
Si l'événement arrive lors de la fermeture de la porte, cette dernière remonte immédiatement jusqu'à la position finale haute.

Pour des raisons de sécurité, au moment où survient l'événement, si la porte se trouve dans un état "indéterminé" (après un déplacement manuel), cette cette dernière s'ouvre jusqu'à la position finale haute.éterminé" (après un déplacement manuel), cette dernière s'ouvre jusqu'à la position finale haute.



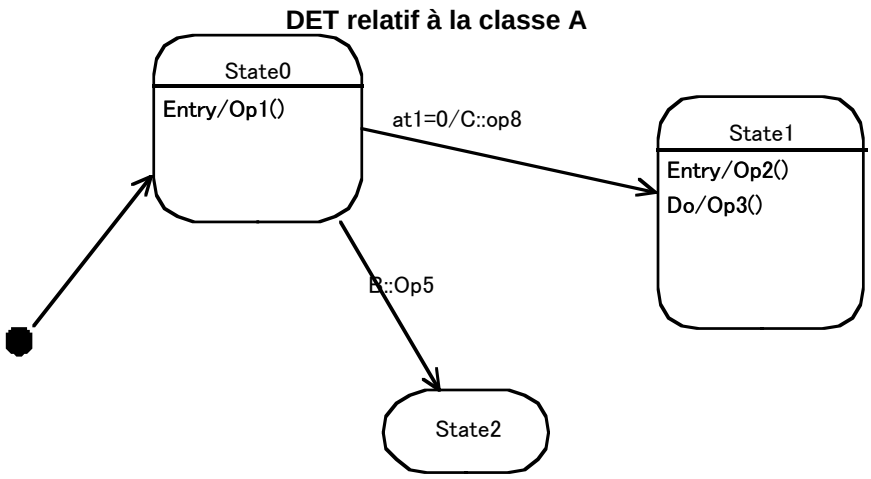
7-12) vers les diagrammes statiques

Un diagramme état transition va permettre, à partir d'un diagramme statique existant, de compléter les propriétés (et éventuellement les méthodes). **Si** les diagrammes d'interaction ont été suffisament complets, il ne devrait pas devoir mettre à jour les associations entre les classes.



7-13) Exercice de synthèse

A partir des diagrammes état-transitions suivants, mettre à jour le diagramme statique.



DET relatif à la classe B

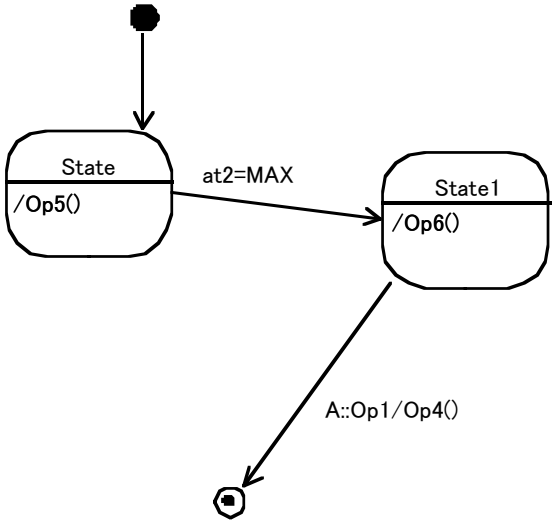
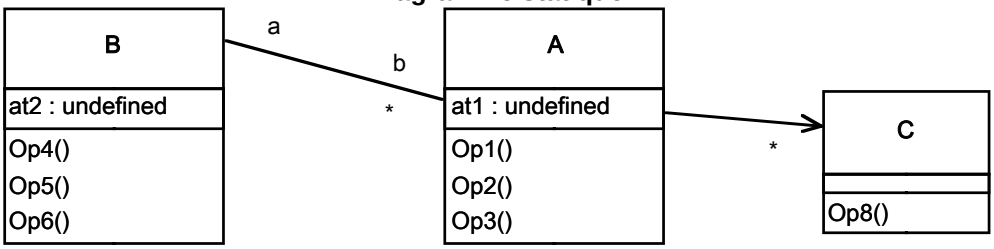


Diagramme statique



8) Les diagrammes d'activités

C'est une variante des diagrammes Etats-transitions. Là, ce sont les activités qui sont mises en avant.

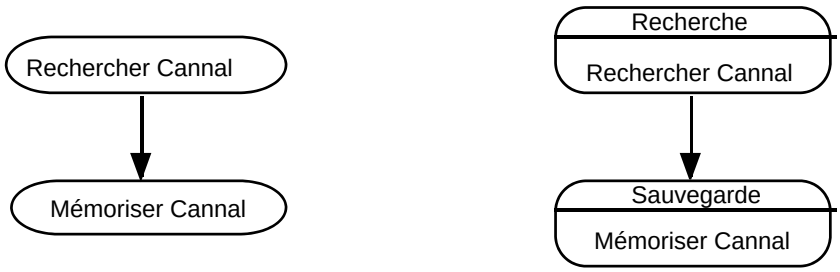
Il modélise le comportement interne d'une méthode (réalisation d'une opération), d'un cs d'utilisation ou d'un procesus.

Son utilisation est intéressante pour décrire des séquences ou des combinaisons d'opérations avec transitions internes ou automatiques. Dans le cas contraire, un diagramme état-transition est préférable.

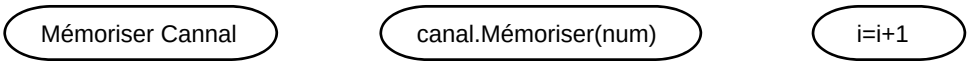
La pluppart des éléments utilisés dans les DET sont utilisables dans les DA.

8-1) les états-action

Un état action est un état simplifié dans lequel figure une action d'entrée et avec au moins une transition automatique vers un autre état. On le représente sous la forme d'un rectangle arrondi légèrement plus convexe que dans un DET.



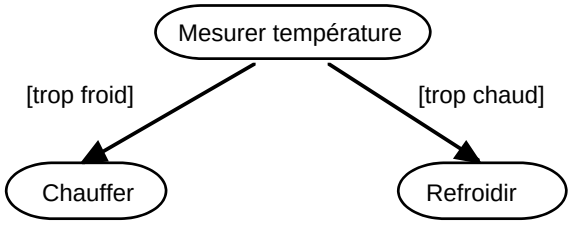
Un état action peut être décrit au moyen du langage naturel, de pseudo-code, de langage de programmation.



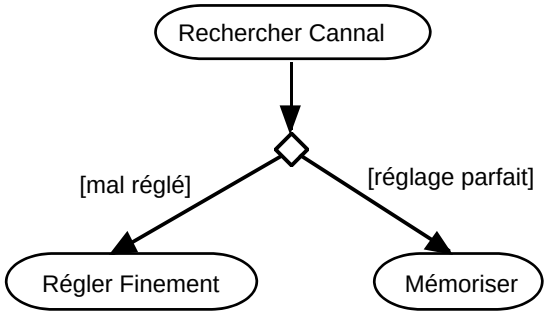
8-2) les transitions

En général, les transitions sont automatiques.
Elles sont représentées par des flèches comme dans les DET
Le déclenchement de l'action suivante est instantanné
Il est inutile de faire figurer le nom de l'événement sur la transition

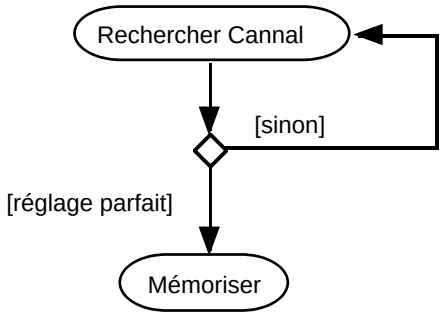
Elles peuvent parfois être déclenchées par un événement particulier.
Des conditions booléennes appelées décisions peuvent parfois conditionnées des transitions mutuellement exclusives. On place les gardes à côté des transitions.



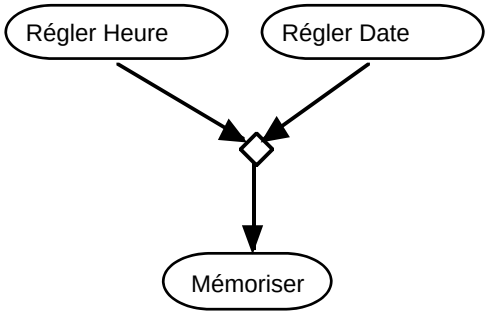
On peut ajouter un losange pour symboliser une transition conditionnelle.



Il existe une garde sinon



Ce même losange peut également servir de point de jonction

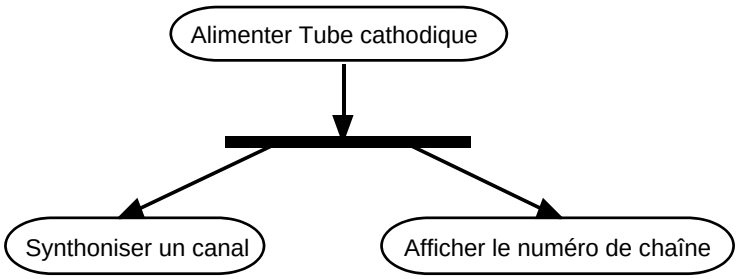


Remarque : Ces diagrammes ci-dessus, nous rappelle, les ordinogrammes utilisés en programmation procédurale. Effectivement, un objet, en interne, peut se comporter comme une petite application procédurale.

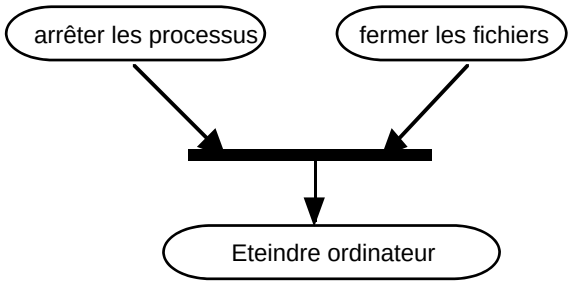
8-3) Synchronisation

On peut représenter les synchronisations entre flots au moyen de transitions concurrentes. Une barre de synchronisation permet d'ouvrir et de fermer des branches parallèles.

Les activations sont simultanées.

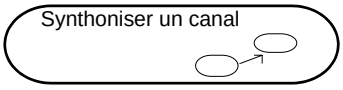


Les déclenchements ne sont eux pas simultanés. La synchronisation sera franchie aux déclenchements de toutes les transitions.



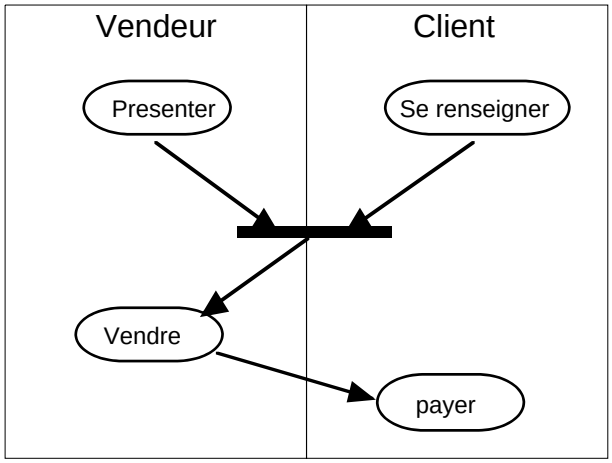
8-4) Etats à sous activités

Chaque état peut se décomposer en diagramme d'activité. Toutes les règles sont les mêmes que pour les DET.



8-5) Les travées

les DA peuvent être découpés en travées pour découper les responsabilités dans un système.
Les travées sont des **couloirs d'activités**
Les transitions peuvent traverser les travées. Des lignes verticales délimitent les travées.



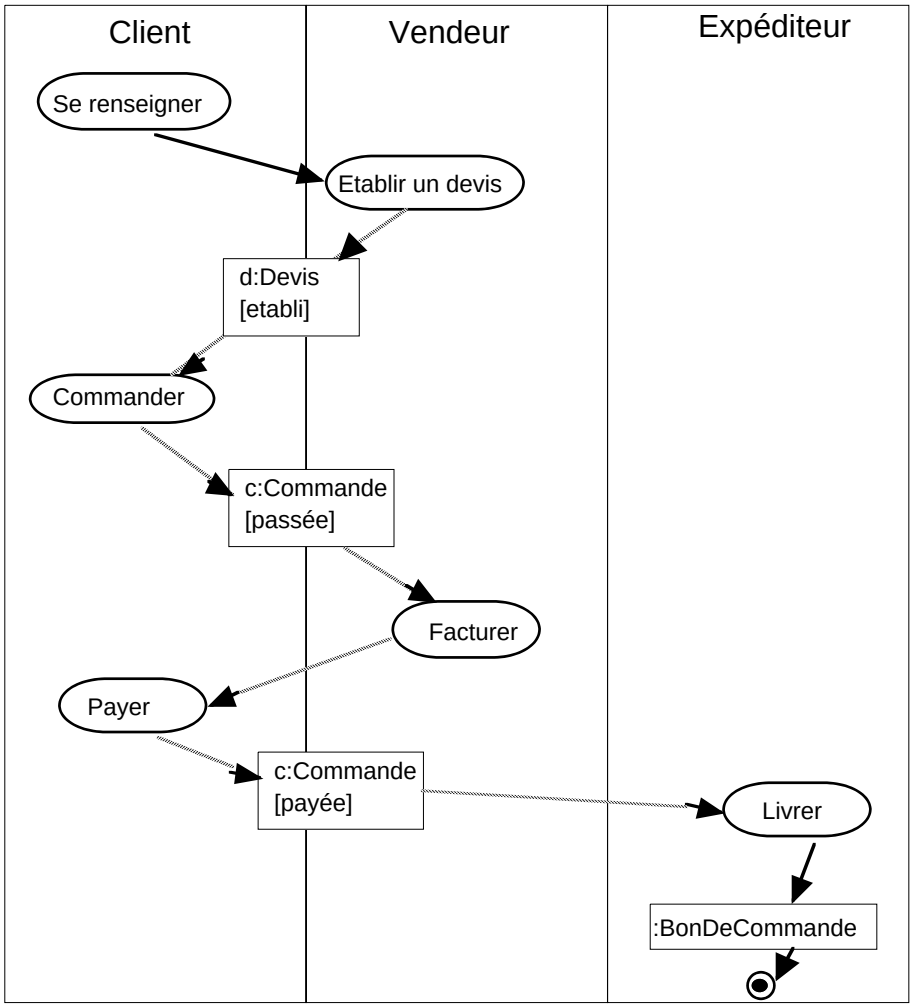
8-6) Flots entre actions et travées

Il est possible de faire apparaitre des objets dans un diagramme d'activité. Ces objets sont ceux qui

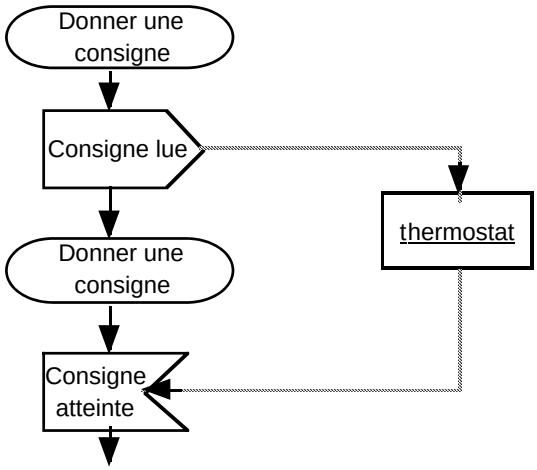
- initient les actions : on connecte l'objet à l'état action associé par une flèche en trait pointillé
- sont utilisés par des actions : on connecte l'objet à l'état action associé par une flèche en trait pointillé
- sont modifiés par des actions : on connecte l'état action à l'objet associé par une flèche en trait pointillé

Un même objet peut être modifié par une action puis utilisé par une autre.

On peut associé un état à l'objet et le représenter plsieurs fois dans le diagramme.



8-7) Icônes associés aux transitions



L'envoi d'un signal se symbolise par un pentgone convexe. Ce symbole peut être relié à l'objet destinataire.

L'attente d'un signal se symbolise par un pentgone concave. Ce symbole peut être relié à l'objet émetteur.

Ce symbole est inséré entre deux états-actions reliés par une transition automatique. La signature du signal est contenue dans le penragone.

9) Les diagrammes de composants

Ils décrivent des composants et leurs dépendances du point de vue réalisation. Ils sont des vues statiques de l'implémentation. Ils montrent les choix de réalisation (qui suit la conception détaillée).

Ce diagramme sert à la documentation du code mais n'intervient pas dans le processus d'analyse et de conception.

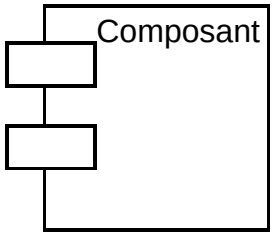
9-1) Les composants

Un composant es un élément physique qui représente une partie implémentée d'un système. Cela peut être :

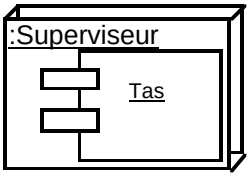
- du code (source, binaire, exécutable)
- un fichier de commande
- un script
- un fichier de données
- une base de données

Il peut être connecté à d'autres composants par des dépandances ou des compositions.
Il peut appartenir à un ou à plusieurs noeuds

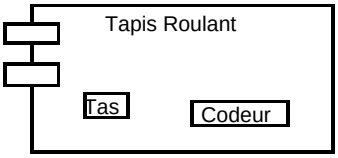
On le repésente par un rectangle principal et par deux petits rectangles sur la partie gauche (symbolise un "programme"). Le nom du composant est placé dans le rectangle principal.



Il est possible de montrer l'instance de noeud dans laquelle une instance de composant est instanciée



Les classes appartenant à un composant peuvent etre représentées dans celui-ci.



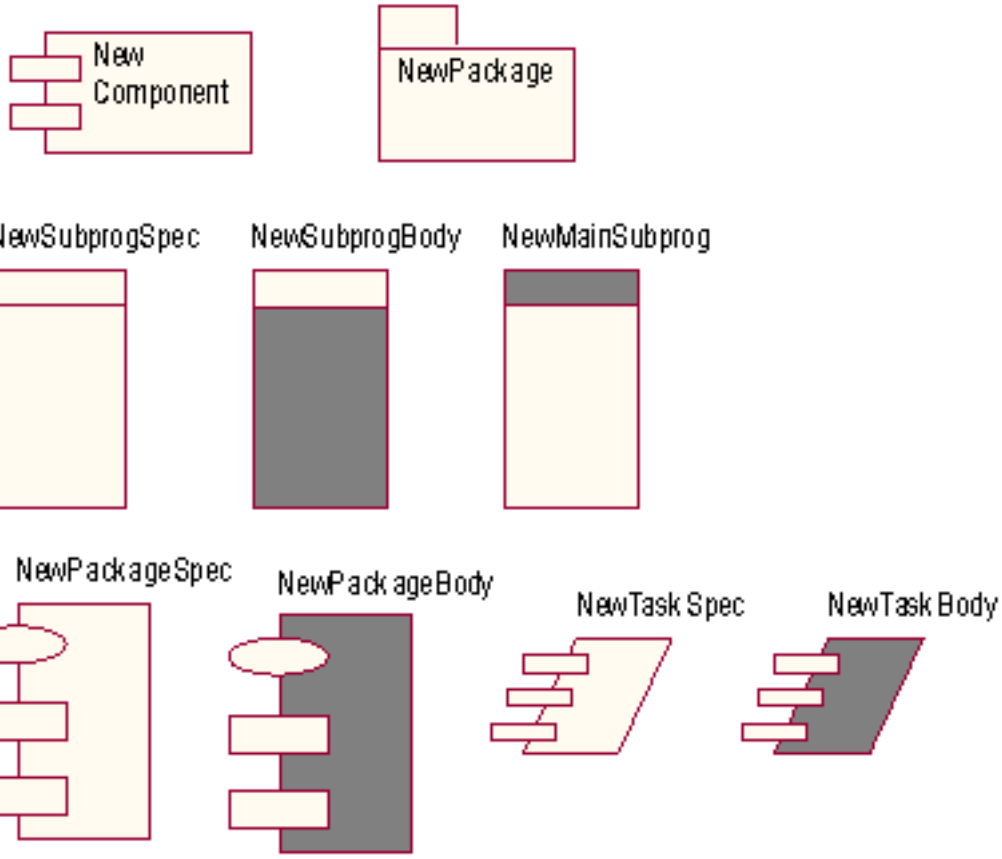
UML définit un certains nombres de prototypes comme :

- <<document>> : un document quelconque
- <<exécutable>>
- <<fichier>> : code source ou données
- <<bibliothèque>>
- <<table>>
- <<processus lourd>> ou <<process>>
- <<processus léger>> ou <<thread>> (rq : on parle aussi de sous tâches ou de fil d'exécution)

9-2) Les modules

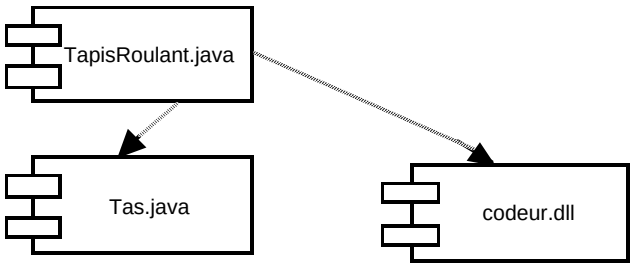
Pour représenter des applications informatiques, on peut utiliser des symboles personalisés que l'on appelle des modules. suivant les langages de programmation, on symbolisera les spécifications, les corps (ou définitions), les patrons, les paquetages ...

Voici la représentation version Rose 98

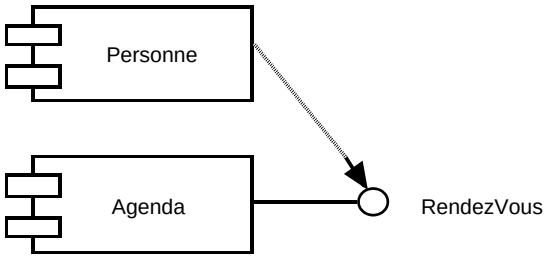


9-3) Les dépendances

Les relations de dépendance sont utilisées dans les diagrammes de composants pour indiquer qu'un composant fait appel aux services d'un autre composant. Une flèche pointillée (que l'on peut nommée dépend de ou utilise ou nécessite) réalise cette association.



Il est également possible de montrer qu'un composant utilise l'inteface d'un autre composant.



10) Les diagrammes de déploiement

Ils montrent la disposition physique des différents matériels (les noeuds) qui composent le système. Ils peuvent montrer la répartition des composants sur ces matériels.

On peut montrer des classes ou des nstances de noeuds.

10-1) Les noeuds

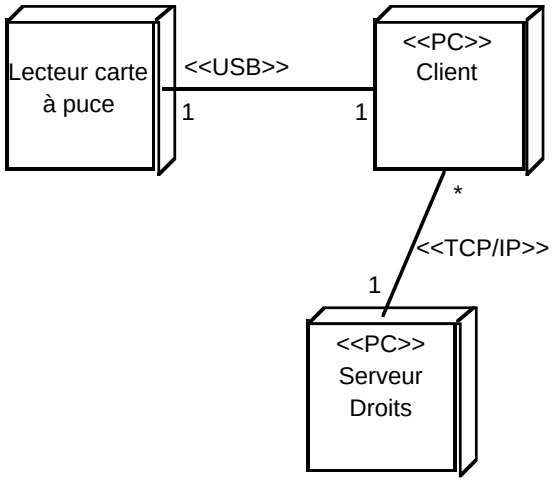
Chaque ressource matérielle est représentée par un noeud. La représentation d'un noeud est un cubes en 3D avec le nom à l'intérieur. Il peut être plus intéressant de remplacer ces cubes par des représentations plus parlantes des matériels.

On peut ajouter des propriétés au noeud (pour indiquer le type de processeur ou le système d'exploitation). Un stéréotype informe de la propriété.

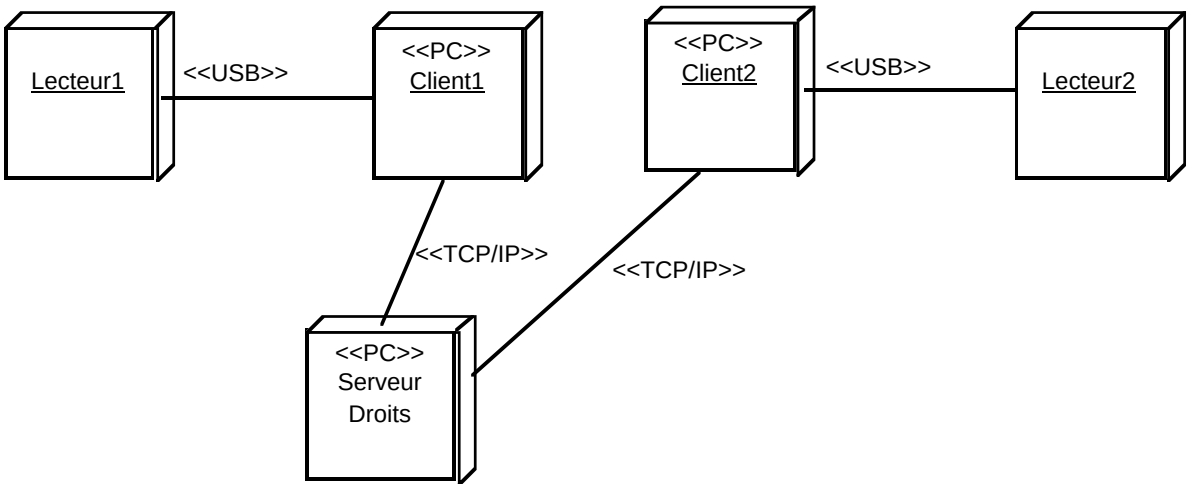
La encore, on peut montrer des classes ou des instances de noeud (même façon de le montrer ...)

10-2) Les supports de communication

Les relations entre les noeuds désignent le type de support communication. Ce dernier peut désigner une couche 1 à 4 (en général) du modèle OSI.

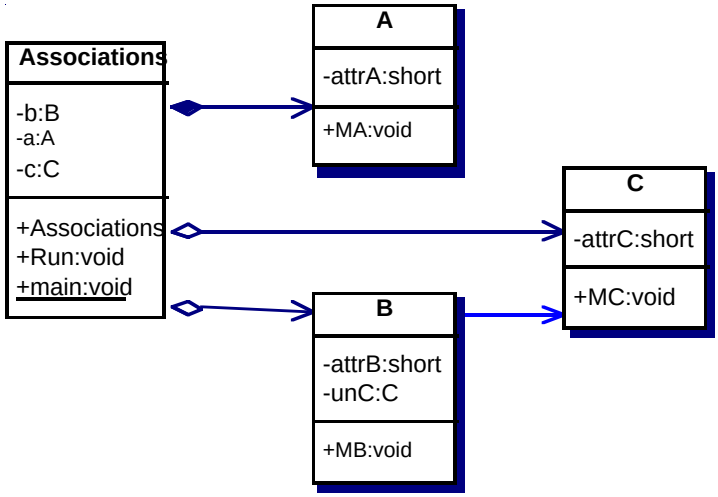


Les informations de multiplicités peuvent être présentes dans les diagrammes de classes de noeud.



11) Classes partculières et implémentation en Java et C++

11-1) Les diverses associations




```java
//version java
class A{
  private short attrA;
  public void MA(){
    attrA=5;
  }
}
class B{
  private short attrB;
  private C unC;
  public void MB(C ceC){
    attrB=6;
    unC = ceC;
    ceC.MC();
  }
}
class C{
  private short attrC;
  public void MC()
  {
    attrC=7;
  }
}
class Associations{
  private B b;
  private A a;
  private C c;
  public Associations(){
    a = new A();
  }
  public void Run(){
    b = new B();
    c = new C();
    b.MB(c);
  }
  public static void main(String[] args){
    Associations assoc = new Associations();
    assoc.Run();
    b = null;
  }
}
```

```cpp
// version C++
class A{
  private:
    short attrA;
  public:
    void MA(){
      attrA=5;
    }
};
class C{
  private:
    short attrC;
  public:
    void MC(){
      attrC=7;
    }
};
class B{
  private:
    short attrB;
    C *unC;
  public:
    void MB(C *ceC){
      attrB=6;
      unC = ceC;
      ceC->MC();
    }
} ;
class Associations
{
  private:
    B *b;
    A a;
    C *c;
  public:
    Associations(){}
    void Run(){
      b = new B();
      c = new C();
      b->MB(c);
      delete b;;
    }
};
void main(int argc, char* argv[]){
    Associations assoc;
    assoc.Run();
}
```

11-2) Les classes utilitaires (voir utilitaire.cpp et bidon.java)

Il est parfois utile de regrouper des éléments (des fonctions d'une bibliothèque par exemple) au sein d'un même module sans pour autant vouloir construire une classe complète. Une classe utilitaire permet de représenter de tels modules et de les manipuler graphiquement comme des classes conventionnelles.

Les classes utilitaires ne peuvent être instanciées. (ce n'est pourtant pas des classes abstraites).

En C++ et en Java, une classe utilitaire est une classe qui ne contient que des membres statiques (opérations et propriétés). L'emploi de ces classes est indispensable en java où l'on ne peut utiliser de données globales (variables, constantes, fonctions ...)

Le stéréotype <<Utilitaire>> spécialise les classes comme utilitaires.

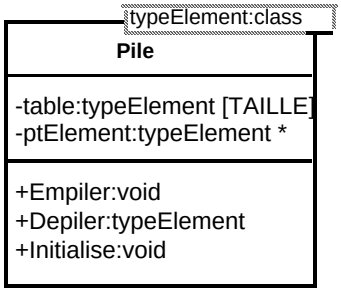
```
// p1225java.java
class Util
{
  public static short VRAI = 1;
  public static short FAUX = 0;
  public static short TAILLE = 10;
}

public class Bidon
{
  private short attr1;
  private int[] tab;
  public static void main(String[] args)
  {
    Bidon bidon = new Bidon();
    bidon.Init();
  }
  Bidon()
  {
    attr1 = Util.VRAI;
    tab = new int[Util.TAILLE];
  }
  public void Init()
  {
    tab[0] = 5;
    System.out.println("coucou");
    try { Thread.sleep(500); // ms
    } catch (InterruptedException e) {}
  }
}
```

```
// p1225cpp.cpp
class Util
{
  public:
    const static short TAILLE = 10;
    const static short VRAI = 1;
    const static short FAUX = 0;
};

class Bidon
{
  private:
    short attr1;
    int *tab;
  public:
    Bidon()
    {
        attr1 = Util::VRAI;
        tab = new int[Util::TAILLE];
    }
    void Init()
    {
        tab[0] = 5;
        cout << "coucou";
        Sleep(500);
    }
};

void main(int argc, char* argv[])
{
    Bidon bidon;
      //ou Bidon *bidon = new Bidon;
    bidon.Init();
}
```

11-3) Les classes paramétrables

Les classes paramétrables sont des modèles de classes. Elles correspondent aux templates de C++ (Patrons). Elle ne peut être utilisée telle quelle. Il convient d'abord de l'instancier afin d'obtenir une classe réelle qui pourra à son tour être instanciée afin de donner un objet.

Lors de l'instanciation, les paramètres effectifs personnalisent la classe réelle obtenue à partir de la classe paramétrable. Ces dernières permettent de construire des collections universelles, typées par les paramètres effectifs.



Les classes paramétrables sont surtout utilisées en conception détaillée.

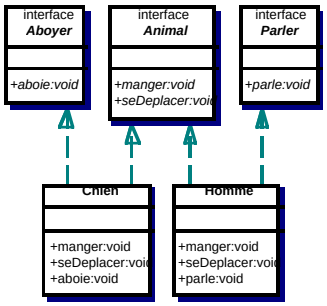
```
// exemple de code en C++
template <class typeElement> class Pile
{
  protected:
          typeElement table[TAILLE];
          typeElement *ptElement;
  public:
          void Empiler(typeElement el)
     {
       ptElement--;
       *ptElement = el;
     }
          typeElement Depiler()
     {
          return *(ptElement++);
     }
          void Initialise()
     {
          ptElement = table + TAILLE;
     }
};

void main()
{
  Pile    <int>          unePileDentier;
  Pile    <float>        unePileDeFlottant;
  int     tab[5];

  unePileDentier.Initialise();
  unePileDentier.Empiler(25);
  unePileDentier.Empiler(34);
  unePileDentier.Empiler(12);
  tab[0]=unePileDentier.Depiler();
  tab[1]=unePileDentier.Depiler();
  tab[2]=unePileDentier.Depiler();

}
```

11-4) Les interfaces

Une interface utilise un type pour décrire le comportement visible d'une classe, d'un composant, ou d'un paquetage.

Les interfaces sont présentées sous la forme d'un petit cercle relié à la classe qui fournit les services On peut également les représenter au moyen de classes stéréotypées.

Une interface fournit une vue totale ou partielle d'un ensemble de services offerts par un ou plusieurs éléments. Les dépendants d'une interface utilisent tout ou partie des services décrits par l'interface.



Il n'y a pas d'implémenation des interfaces pour la langage C++

Dans Java, les interfaces sont un type particulier de classes, qui permettent à une autre classe d'hériter un comportement auquel elle n'aurait pas accès autrement. Elles définissent des ensembles de méthodes implémentées par les classes du programme.Il faut savoir que la hiérarchisation Objet de Java fait qu'une classe ne peut hériter que d'une seule classe, et non plusieurs. Il remplace la notion d'héritage multiple, utilisée en C++. Surtout, les interfaces servent à créer des comportements génériques: si plusieurs classes doivent obéir à un comportement particulier, on créé une interface décrivant ce comportement, on est la fait implémenter par les classes qui en ont besoin. Ces classes devront ainsi obéir strictement aux méthodes de l'interface (nombre, type et ordre des paramètres, type des exceptions), sans quoi la compilation ne se fera pas.Les interfaces sont très similaires aux classes (une interface par fichier .class...), à la différence que les interfaces ne peuvent être instanciées

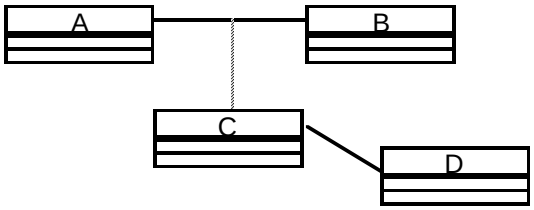
```
// http://developpeur.journaldunet.com/tutoriel/jav/031029jav_interfaces1.shtml
```

```
// exemple de code en Java
interface Animal{
  // tous les animaux doivent implémenter les méthodes suivantes
  void manger(String nourriture);
  void seDeplacer();
}
interface Parler{
  void parle(int phrase);
}
interface Aboyer{
  void aboie();
}

class Homme implements Animal, Parler{
  //implémentations de l'interface Animal
  public void manger(String nourriture)
  {
    System.out.println("Miam, " +nourriture+ "!");
  }
  public void seDeplacer()
  {
    System.out.println("déplacement de l'homme");
  }
  //implémentations de l'interface Parler
  public void parle(int phrase)
  {
    System.out.println(phrase);
  }
}
class Chien implements Animal, Aboyer{
  //implémentations de l'interface Animal
  public void manger(String patee)
  {
    System.out.println("Miam, " +patee+ "!");
  }
  public void seDeplacer()
  {
    System.out.println("déplacement du chien");
  }
  //implémentations de l'interface Aboyer
  public void aboie()
  {
    System.out.println("Ouaf!");
  }
}
public class InterfaceJava{
  public InterfaceJava()
  {
    Chien viensMonChien = new Chien() ;
    Homme pepere = new Homme();
    viensMonChien.manger("desRestes");
    pepere.manger("desVachesFolles");
  }
  public static void main(String arg[])
  {
    InterfaceJava monInter;
    monInter = new InterfaceJava();
  }
}
```

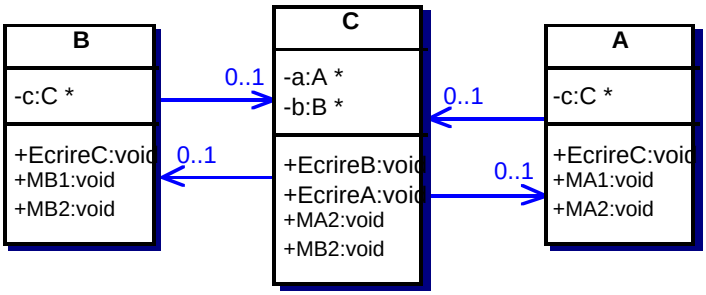
11-5) Les classes associations

Une association peut être représentée par une classe pour ajouter, par exemple des attributs et des opérations dans l'association. Une classe de ce type, est une classe comme une autre et peut participer à d'autres associations dans le système.

La notation utilise un trait pointillé pour attacher une classe à une association. Dans l'exemple suivant, l'association entre les classes A et B est représentée par la classe C, qui elle même est associée à la classe D.



Une association qui contient des attributs sans participer à des relations avec d'autres classes est appelée association attribuées. Elle ne porte pas de nom spécifique dans ce cas.

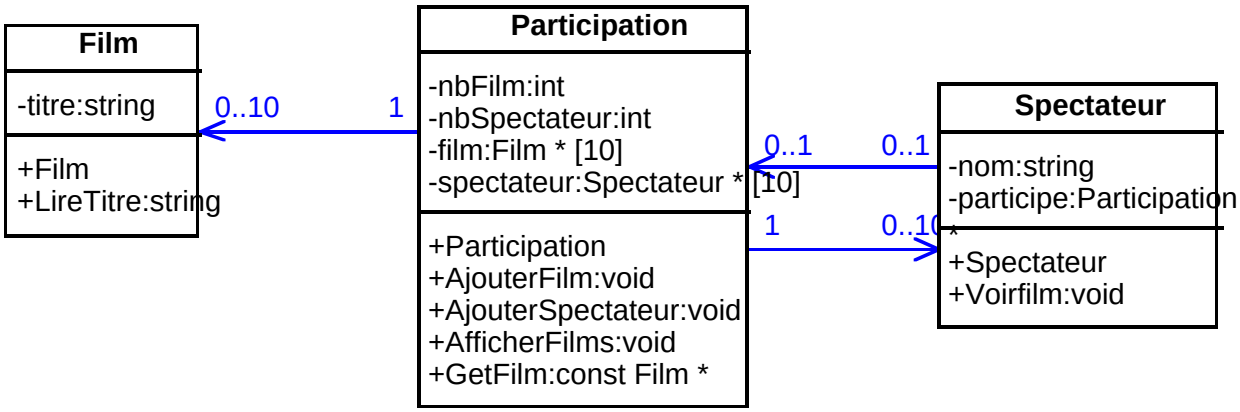



```
// version java
class C // c'est la classe association
{
    private A a;
    private B b;
    public void EcrireB(B unB){b = unB;}
    public void EcrireA(A unA){a = unA;}
    public void MA2(){  a.MA2(); }
    public void MB2(){  b.MB2(); }
}

class A
{
    private C c;
    public void EcrireC(C  unC){c = unC;}
    public void MA1() {   c.MB2(); }
    public void MA2(){}
}
class B
{
    private C c;
    public void EcrireC(C  unC){c = unC;}
    public void MB1() {  c.MA2();  }
    public void MB2(){}
}

public class ClaAssociation
{
  private C monC;
  private A monA;
  private B monB;

  public static void main(String arg[])
  {
    ClaAssociation claAssociation;
    claAssociation = new ClaAssociation();
    claAssociation.Run();
  }
  void Run()
  {
    monC = new C();
    monA = new A();
    monB = new B();
    monA.EcrireC(monC);
    monB.EcrireC(monC);
    monC.EcrireA(monA);
    monC.EcrireB(monB);

    monA.MA1();
    monB.MB1();
  }
}
```

```
// version C++
class A;class B;

class C // c'est la classe association
{
    private:
        A *a;
        B *b;
    public:
        void EcrireB(B *unB){b = unB;}
        void EcrireA(A *unA){a = unA;}
        void MA2();
        void MB2();
};

class A
{
    private:
        C *c;
    public:
        void EcrireC(C * unC){c = unC;}
        void MA1();
        void MA2();
};
class B
{
    private:
        C *c;
    public:
        void EcrireC(C * unC){c = unC;}
        void MB1();
        void MB2();
};

void C::MA2()
{
  a->MA2();
}
void C::MB2()
{
  b->MB2();
}
void A::MA1()
{
  c->MB2();
}
void A::MA2(){}
void B::MB1()
{
  c->MA2();
}
void B::MB2(){}


void main()
{
    C monC;
    A monA;
    B monB;

    monA.EcrireC(&monC);
    monB.EcrireC(&monC);
    monC.EcrireA(&monA);
    monC.EcrireB(&monB);

    monA.MA1();
    monB.MB1();
}
```

Note : les classes d'association sont surtout utiles lorsque l'on associe des classes avec des cardinalités supérieures à 1..1

Exemple : plusieurs spectateurs voient plusieurs films ...




```
// participation.h
#include "spectateur.h"

class Participation
{
    private:
        int nbFilm;
        int nbSpectateur;
        Film *film[10];
        Spectateur *spectateur[10];
    public:
        Participation();
        void AjouterFilm(Film *); // association avec plusieurs films
        void AjouterSpectateur(Spectateur *); // et plusieurs spectateurs
        void AfficherFilms();
        const Film * GetFilm(int);
};
#else
class Participation;
#endif

// participation.cpp
#include "participation.h"
#include <iostream>
using namespace std;

//-------------------------------------------------------------------------
Participation::Participation()
{
    nbFilm = 0;
    nbSpectateur = 0;
}
//-------------------------------------------------------------------------
void Participation::AjouterFilm(Film *_film)
{
    film[nbFilm++] = _film;
}
//-------------------------------------------------------------------------
void Participation::AjouterSpectateur(Spectateur *_spectateur)
{
    spectateur[nbSpectateur++] = _spectateur;
}
//-------------------------------------------------------------------------
void Participation::AfficherFilms()
{
```

```
    int i=0;
    while (i< nbFilm)
    {
        cout << i << " : " << film[i]->LireTitre() << endl;
        i++;
    }
}
//---------------------------------------------------------------------------
const Film * Participation::GetFilm(int num)
{
    return film[num];
}
//---------------------------------------------------------------------------
```

```
// spectateur.h
#ifndef spectateurH
#define spectateurH
#include <string>
using namespace std;
#include "participation.h"

class Spectateur
{
    private:
        string nom;
        Participation *participe;
    public:
        Spectateur(string, Participation *);
        void Voirfilm(int);
};
#else
class Spectateur;
#endif
```

```
// spectateur.cpp
#include "spectateur.h"

//---------------------------------------------------------------------------
Spectateur::Spectateur(string _nom, Participation *_participation)
{
    nom = _nom;
    participe = _participation;
}
//---------------------------------------------------------------------------
void Spectateur::Voirfilm(int num)
{
    participe->GetFilm(num)->Visionner();
}
//---------------------------------------------------------------------------
```

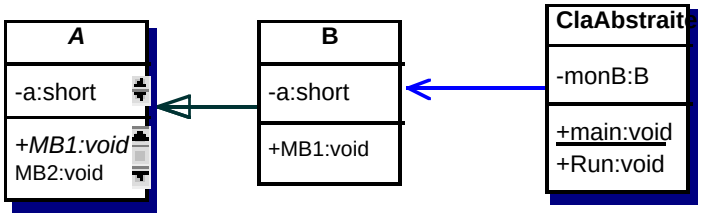
```
// film.h
#ifndef filmH
#define filmH
#include <string>
using namespace std;
#include "participation.h"

class Film
```

```
{
    private:
        string titre;
      //  Participation *participe; // pas nécessaire si unidirectionnel
    public:
        Film(string, Participation *);
        string LireTitre();
        void Visionner();
};
#else
class Film;
#endif

// film.cpp
#include "film.h"
#include <iostream>
using namespace std;

//-------------------------------------------------------------------------
Film::Film(string _titre, Participation *_participation)
{
    titre = _titre;
//    participe = _participation;  // pas nécessaire si unidirectionnel
}
//-------------------------------------------------------------------------
string Film::LireTitre()
{
    return titre;
}
//-------------------------------------------------------------------------
void Film::Visionner()
{
    cout << "démarrage du film " << titre << endl;
}

// appli.cpp
#include "participation.h"
int main(int argc, char* argv[])
{
    Participation participation;
    Film film1("Enfermé dehors",&participation);
    Film film2("Le gout des autres",&participation);
    Film film3("le retour du jedi",&participation);
    Film film4("Indigène",&participation);
    Spectateur spectateur1("denis",&participation);
    Spectateur spectateur2("mathias",&participation);
    Spectateur spectateur3("nicolas",&participation);
    Spectateur spectateur4("eric",&participation);

    participation.AjouterFilm(&film1);
    participation.AjouterFilm(&film2);
    participation.AjouterFilm(&film3);
    participation.AjouterFilm(&film4);
    participation.AfficherFilms();
    spectateur1.Voirfilm(1);
    spectateur1.Voirfilm(2);
    spectateur2.Voirfilm(2);

    return 0;
}
```

11-6) Les classes abstraites

Les classes abstraites ne sont pas directement instanciables; elles ne donnent pas naissance à des objets, mais servent de spécification plus générale (modèles) pour manipuler des objets dérivés de celles-ci. Une classe est désignée abstraite en ajoutant abstraite à son nom ou en mettant le nom en italique.

On peut également appliquer la propriété abstraite à une opération




```
// exemple en java
abstract class A
{
    private short a;
    abstract public void MB1();
    void MB2() { a--;}
}
class B extends A
{
    private short a;
    public void MB1() {  MB2();  }
}

public class ClaAbstraite
{
  private B monB;
  public static void main(String arg[])
  {
    ClaAbstraite claAbstraite;
    claAbstraite = new ClaAbstraite();
    claAbstraite.Run();
  }
  public void Run()
  {
    monB = new B();
    monB.MB1();
  }
}
```

```
// exemple en C++
class A
{
    private:
      short a;
    public:
      virtual void MB1()=0;
      void MB2() { a--;}
};
class B: public A
{
    private:
      short a;
    public:
      void MB1() {  MB2();  }
};
int main(int argc, char* argv[])
{
  B monB;
  monB.MB1();
  return 0;
```