UML : QUELQUES CONSEILS PRAGMATIQUES

Septembre 2020

Modélisation Objet – UML
Stéphane Penhoët
Université d'Evry – Licence informatique L3

Table des matières

1 Introduction 4
2 Diagrammes de classe 5
2.1 Les multiplicités 5
2.1.1 De quel côté se place la multiplicité ? 5
2.1.2 Quand choisir 0..* ou 1..* 6
2.1.3 Les multiplicités des relations ternaires 6
2.1.4 Les multiplicités « automatiques » 7
2.2 Association, agrégation ou composition ? 8
2.2.1 Exemples 9
2.2.2 Considérations pratiques 9
2.3 Rôle ou héritage ? 10
2.3.1 Héritage 10
2.3.2 Rôle 10
2.3.3 Choisir entre héritage et rôle 11
2.4 Rôle ou état ? 13
2.5 La transitivité 15
3 Cas d'utilisation 16
3.1 Granularité 16
3.2 Include, Extend 17
3.2.1 Représentation du lien 17
3.2.2 Quand faut-il lier les cas d'utilisations entre eux ? 17
3.3 Proposition de démarche 20
4 Diagramme d'états-transitions 21
4.1 Ce qu'il faut savoir sur les transitions 21
4.1.1 Syntaxe 21
4.1.2 Il faut pouvoir sélectionner une transition sortante 22
4.2 Plusieurs états simultanés ? 22
4.2.1 Etats totalement indépendants 23
4.2.2 Etats temporairement indépendants 24
4.3 Considérations pratique 27
5 Diagrammes de séquence 28
5.1 Modélisation de la recherche 28
5.1.1 Problématique 28

5.1.2 Ce qu'il ne faut pas faire 28
5.1.3 Pattern 1 : la recherche portée par la classe de l'objet cherché 29
5.1.4 Pattern 2 : la recherche portée par une classe « gestionnaire d'ensemble » 30
5.1.5 Considérations pratiques 30

1 Introduction

Ce document vise à aborder certaines difficultés souvent rencontrées lorsqu'on modélise en UML.

Il est construit comme un ensemble de recettes qu'on va pouvoir appliquer, tantôt pour trouver comment modéliser tel ou tel cas, tantôt pour vérifier que la modélisation est cohérence. Les chapitres reprennent les différents types de diagrammes vus lors des travaux dirigés. Les paragraphes, quant à eux, adressent chacun une difficulté particulière.

Contrairement à un cours dont le contenu est porté par la norme, certaines des affirmations de ce document peuvent relever de l'opinion plus que du fait. Dans le cadre des travaux dirigés, il est possible de prendre tout ce qui est écrit dans le document à la lettre. Dans un contexte professionnel, il est possible que les pratiques locales soient en contradiction. C'est normal. Dans ce cas, il faut s'en tenir à l'adage « à Rome on fait comme les romains ».

Enfin, il se peut que vous trouviez le plan déséquilibré. C'est sans doute parce que c'est le cas :

- Nous avons lourdement insisté sur les diagrammes de classe et de cas d'utilisation parce qu'il s'agit des diagrammes les plus importants et plus utilisés ;
- Nous avons aussi insisté sur le diagramme de séquence, car il s'agit d'un sujet difficile.
- Nous n'abordons que rapidement le diagramme d'états-transition, car il n'y a finalement que deux sujets qui semblent poser des difficultés récurrentes
- Rien n'est dit à ce stade sur le diagramme d'activité, car il ne pose généralement pas de souci particulier.

2.1 Les multiplicités

Les multiplicités représentent le nombre d'objets (instances d'une classe) qu'on pourra avoir dans le cadre d'une association.

2.1.1 De quel côté se place la multiplicité ?

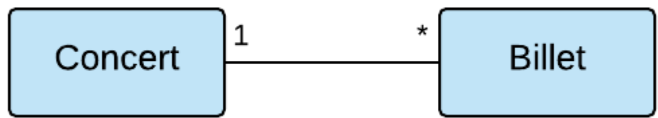
Plutôt que fournir une définition, on va ici voir deux moyens pour ne pas se tromper :

- En retenant un exemple, qu'on pourra invoquer à chaque doute ;
- En appliquant une méthode un peu mécanique.

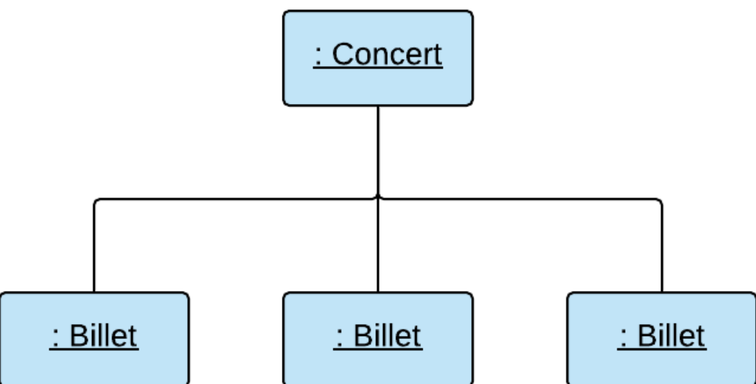
On détaille les deux options ci-dessous.

2.1.1.1 Par l'exemple

La méthode la plus simple pour ne pas se tromper est de mémoriser un exemple. A titre personnel, j'utilise une association entre une classe « concert » et une classe « billet ». Pour un concert il peut y avoir de nombreux billets alors qu'un billet ne correspond qu'à un seul concert. Ce qui donne :



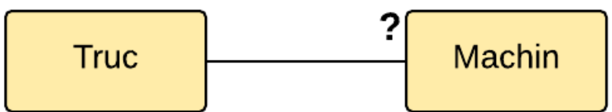
Ce diagramme de classe pour donner lieu – par exemple – au diagramme objet suivant :



Au moindre doute, je peux me remémorer cet exemple et ainsi savoir comment positionner les multiplicités.

2.1.1.2 Par la méthode

Il est aussi possible de voir les choses de façon plus opératoire :



Pour chaque relation on peut se placer d'un côté (ici, du côté « truc »). On se pose alors la question « pour un truc donné, combien puis-je avoir de machins ? ». La réponse doit être écrite au niveau du point d'interrogation. A titre d'exercice, en appliquant cette méthode sur la relation « concert – billet » vous devriez retrouver les multiplicités évoquées.

Il faut ensuite se placer de l'autre côté et se poser exactement la même question… et recommencer pour chacune des associations.

2.1.2 Quand choisir 0..* ou 1..*

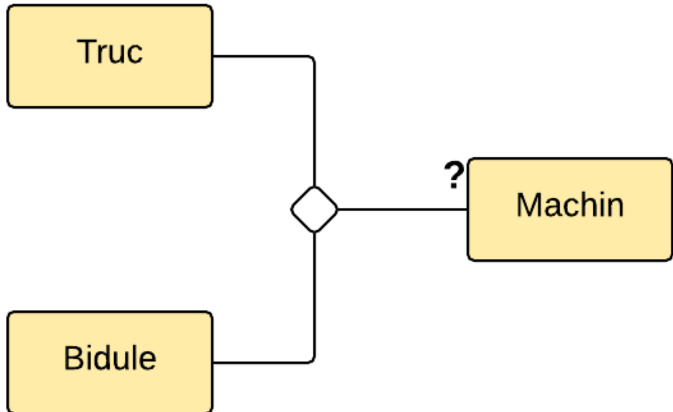
Pour commencer, un rappel. Mettre une multiplicité « * » est synonyme de « 0..* ».

D'une manière générale, quand un modèle n'est pas lié à des aspects techniques, l'expérience montre que : mettre 0 ou 1 est rarement signifiant.

Si on reprend l'exemple du modèle pour du §2.1.1.1 entre un concert et billet :

- On n'a aucune idée du nombre de billets qu'on aura. Ça dépend de la taille de salle. Ça peut rapidement être plusieurs centaines voire plusieurs milliers. La borne inférieure n'est vraiment pas un sujet !
- L'argument « on crée le concert avant de créer les billets » est valide, mais uniquement d'un point de vue implémentation. Fonctionnellement, cet argument n'a que peu d'intérêt.

2.1.3 Les multiplicités des relations ternaires



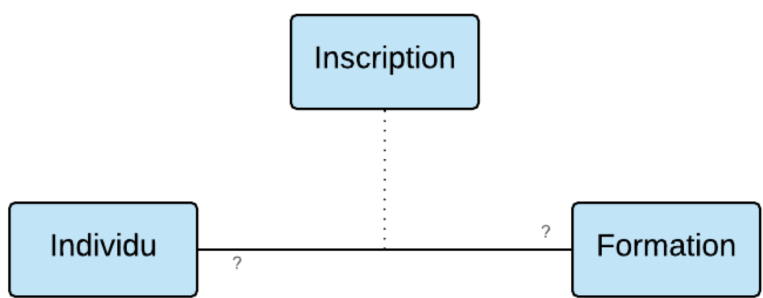
Pour chaque couple (« truc », « bidule »), le point d'interrogation représente le nombre de « machin » qu'il peut y avoir.

2.1.4 Les multiplicités « automatiques »

Certains types d'association vont naturellement restreindre les multiplicités possibles. On détaille dans ce paragraphe les quelques cas qui peuvent se présenter à vous.

2.1.4.1 Classe-association

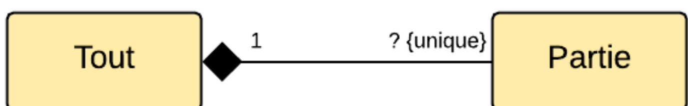
Un classe association va qualifier l'association entre deux autres classes. D'expérience, ce type d'association n'est généralement nécessaire lorsque les multiplicités des deux côtés sont multiples (c'est-à-dire que la multiplicité maximale n'est pas « 1 »).



Dans l'exemple ci-dessus, les multiplicités côté individu comme du côté formation seront « * » ou « 1..* ».

2.1.4.2 Composition

Prenons l'exemple suivant :



Dans le cas d'une composition, le cycle de vie de la « partie » est totalement lié au cycle de vie du tout. De fait, une « partie » donnée ne peut pas être dans deux « tout » différents.

Par exemple, si votre « tout » est un livre et votre « partie » une page, le fait qu'une page soit dans un livre implique qu'elle n'est pas dans un autre.

Cela impose une multiplicité « 1 » côté « tout » (à côté du losange). Il n'y a pas le choix.

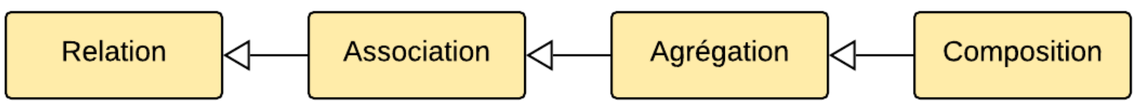
En revanche, la multiplicité côté « partie » peut varier. Dans la pratique, il s'agit souvent de « * », mais ce n'est pas systématique.

Enfin, chaque « partie » est unique. Une page donnée n'est pas deux fois dans un livre.

Dans la pratique, la multiplicité côté losange est rarement indiquée (car implicite). La contrainte « unique » côté « partie » encore moins, les contraintes de ce type étant assez rares.

2.2 Association, agrégation ou composition ?

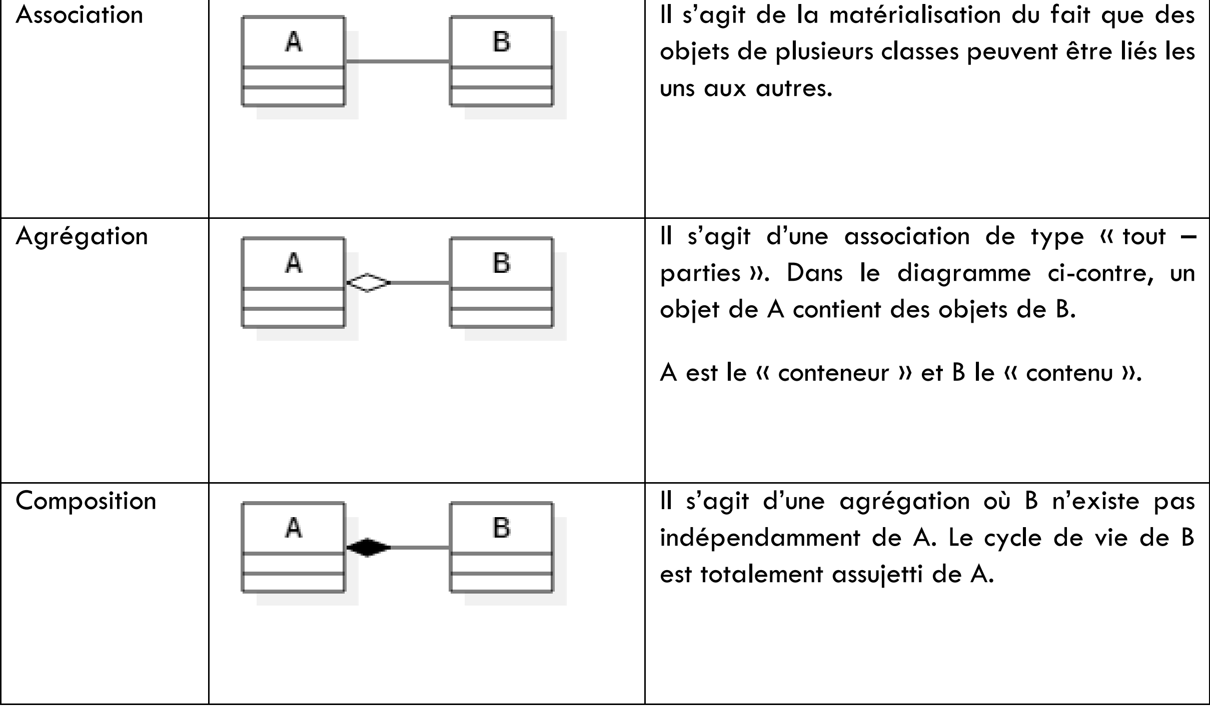
Le métamodèle UML concernant les 3 concepts est le suivant :



Les agrégations sont donc des cas particuliers d'associations, de même que les compositions sont des cas particuliers d'agrégations.

Est une relation tout lien entre deux artefacts en UML. Cet artefact peut-être une classe, un objet, un cas d'utilisation, etc.

Pour ce qui est des trois autres concepts, voici ce qui les caractérise :



Association	A — B	Il s'agit de la matérialisation du fait que des objets de plusieurs classes peuvent être liés les uns aux autres.
Agrégation	A ◇— B	Il s'agit d'une association de type « tout – parties ». Dans le diagramme ci-contre, un objet de A contient des objets de B. A est le « conteneur » et B le « contenu ».
Composition	A ◆— B	Il s'agit d'une agrégation où B n'existe pas indépendamment de A. Le cycle de vie de B est totalement assujetti de A.

2.2.1 Exemples

Classe A	Classe B	Type d'association
Livre	Page	Composition. Nous sommes bien sur une relation de type tout-partie. Qui plus est, si on détruit le livre, on détruit les pages.
Cours	Elève	Plutôt association, vu que cette relation serait une « inscription ». L'agrégation est ici acceptable, si on considère que les élèves sont une composante d'un cours. La composition est une erreur : les élèves existent indépendamment du cours.
Immeuble	Etage	Composition. Plus d'immeuble => plus d'étages.
Voiture	Roue	Plutôt agrégation. La roue peut avoir été fabriquée avant d'avoir été mise sur la voiture. Si on ne s'intéresse pas à ça, la composition est acceptable. L'association n'est pas fausse, mais un peu faible.

2.2.2 Considérations pratiques

Lorsque vous faites un diagramme de classe pour concevoir une implémentation – que vous êtes proches du code, donc – le choix peut avoir une influence certaine. L'agrégation « A inclut B » peut se traduire par exemple, dans le code par un « tableau d'objets B » comme attribut de A. De la même façon, la synchronisation des cycles de vie peut se traduire par du code détruisant les B quand vous détruisez A. A ce niveau, vous pouvez vous appuyez sur ce que vous savez de l'implémentation qui va résulter de votre choix pour faire le bon choix.

Sur un diagramme plus abstrait (ceux qu'on fait dans les cahiers de charges, ou pour illustrer la compréhension du monde réel), il n'y a pas de traduction technique. Le choix du lien n'est alors que porteur d'un « sens », sans conséquence autre. Il est plus facile alors de se tromper, mais une erreur est moins grave. C'est pour cette raison qu'en cas de doute il y a une excellente stratégie :

Si vous n'arrivez pas à déterminer la nature d'une association, choisissez toujours le concept le plus général.

Autrement dit : si vous hésitez entre une association et une agrégation, prenez l'association. En effet : choisir une association là où une agrégation aurait été mieux, ce n'est pas assez précis, alors que je choisir une agrégation là où il fallait choisir une association, c'est commettre une erreur.

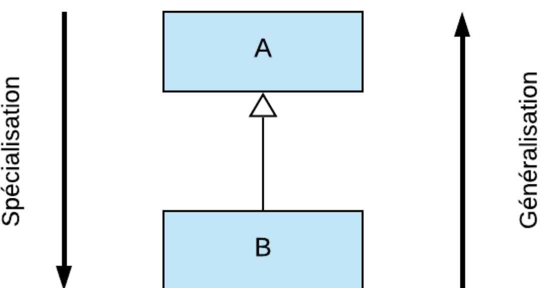
Dernier point important : l'écrasante majorité des associations dans les modèles que l'on rencontre en entreprise sont des associations simples. Le temps que nous avons consacré à ce sujet en cours et en TD est lié au fait que la différence est parfois subtile et nécessite donc pas mal d'explications. Il n'est pas le reflet de l'importance de cette catégorisation.

2.3 Rôle ou héritage ?

2.3.1 Héritage

Lorsqu'il y a un lien d'héritage entre deux classes, cela veut dire qu'une des deux classes possède toutes les caractéristiques d'une autre. D'un point de vue sémantique, on considère que la classe qui hérite des caractéristiques est un cas particulier.

Dans le schéma ci-dessous, nous avons une relation d'héritage entre la classe A et le classe B.



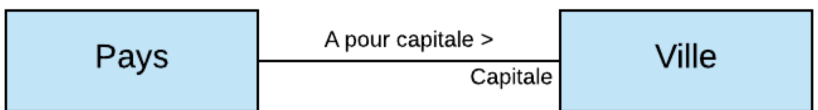
On dit souvent que :

- A est la classe-mère et B la classe fille
- A est la généralisation de B, et B la spécialisation de A
- A est une super-classe et B une sous-classe

2.3.2 Rôle

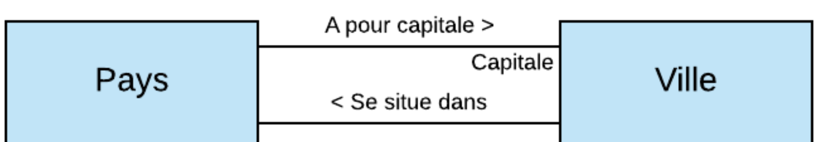
Le « rôle » se définit dans le cadre d'une association entre deux classes. Le rôle d'une classe spécifie que la classe représente dans la relation.

Dans l'exemple ci-dessous, on modélise le fait qu'une ville donnée est la capitale d'un pays :



La notion de ville existe à part entière. Il aurait donc été incorrect de relier le « Pays » à une classe « Capitale ». La notion de Capitale n'existe que du fait de cette relation.

Si, sur le même diagramme, nous souhaitons modéliser le fait qu'un pays contient des villes dont une et une seule est capitale, cela peut simplement se faire en indiquant les deux associations :



2.3.3 Choisir entre héritage et rôle

Il arrive parfois qu'il soit difficile de déterminer s'il faut utiliser un lien de type héritage ou de type rôle pour représenter le lien qui unit deux concepts. L'expérience montre que souvent, dans ce cas, c'est le rôle qu'il faut privilégier.

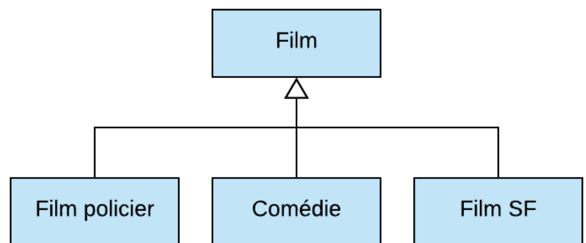
En fait, il arrive même qu'on utilise un héritage par erreur. Heureusement, il existe une méthode pour détecter ce type de situation :

- Il faut « faire tourner » le modèle sur un cas concret. C'est-à-dire établir un diagramme d'objet qui pourrait en être issu ;
- Ce faisant, il faut vérifier qu'on n'instancie pas plusieurs objets pour une entité du monde réel. Si c'est le cas, c'est l'indice d'un problème.

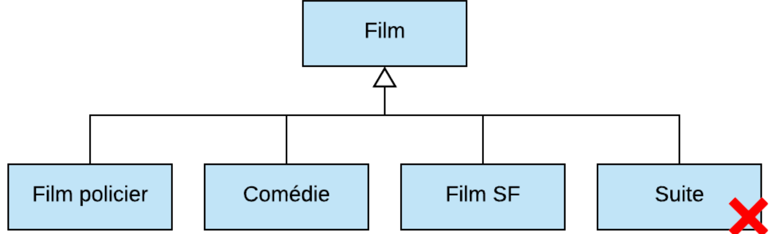
Pour clarifier cette méthode, appliquons-là à un cas simple. Prenons l'énoncé suivant :

Il y a des films policiers, des comédies et des films des science-fiction. Certains films sont des suites d'autres films.

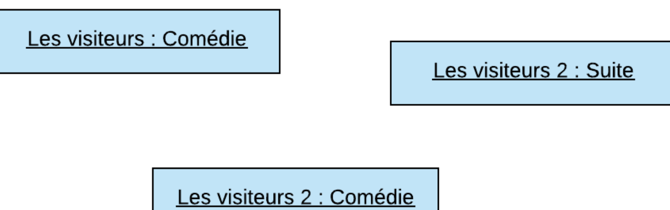
La première phrase est traduite comme suit :



Puis, le modélisateur estime que le fait d'être une suite est aussi une spécialisation de « Film », et ajoute de fait une nouvelle classe-fille (ce qui est une erreur) :

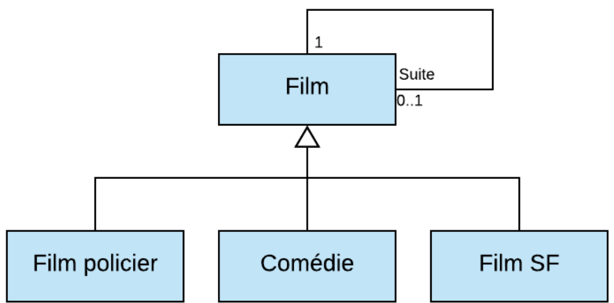


Imaginons qu'il veuille maintenant faire un diagramme d'objet dans lequel il souhaite représenter les films « Les Visiteurs » et « Les Visiteurs 2 ». Cela donnerait quelque chose de la sorte :



Comme le film « Les visiteurs 2 » est à la fois une comédie et une suite, le choix du modélisateur implique que le film génère des objets dans le diagramme d'objet. Ce n'est pas normal : il s'agit d'un film unique. Si c'est une seule entité « dans la vraie vie », cela ne devrait donner naissance qu'à un seul objet dans le modèle objet.

L'erreur du modélisateur est ici de ne pas avoir noté que « *certains films sont des suites **d'autres films*** ». Ce qui est ici surligné en gras devrait être l'indice que nous sommes faces à une relation. Le concept de suite ne se définit que dans le cadre d'une relation entre un film et un autre. Un modèle correct serait donc ici :

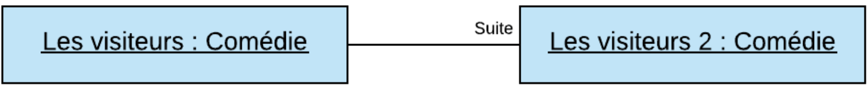


Nous avons intégré un lien auto-référent du film sur lui-même. Le choix des multiplicités est lié au fait que :

- Un film peut ne pas avoir de suite ;
- On considère qu'un film ne peut être que la suite de son prédécesseur.

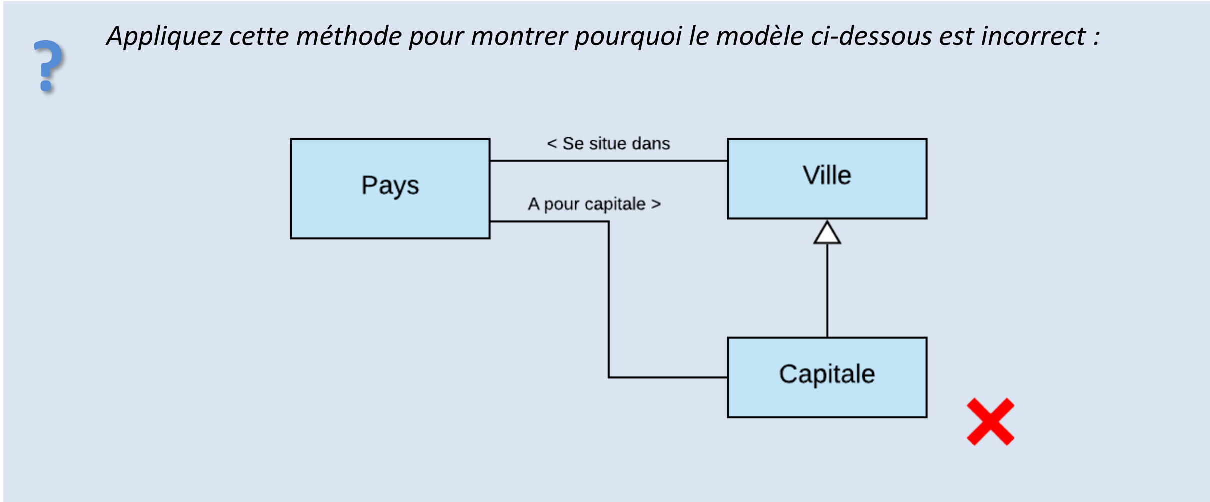
Si au contraire de ces hypothèses vous décidez que « Les Visiteurs 3 » est la suite de « Les Visiteurs » et « Les Visiteurs 2 », alors il vous faudra adapter les multiplicités. Nous aurions aussi pu choisir de nommer la relation « est la suite de » (et l'orienter).

Par acquis de conscience, on peut faire le diagramme d'objet de nos deux films :



Non seulement un objet du monde réel se traduit par un seul objet du modèle, mais en plus nous mettons en évidence le lien entre les deux. C'est beaucoup mieux.

Appliquez cette méthode pour montrer pourquoi le modèle ci-dessous est incorrect :



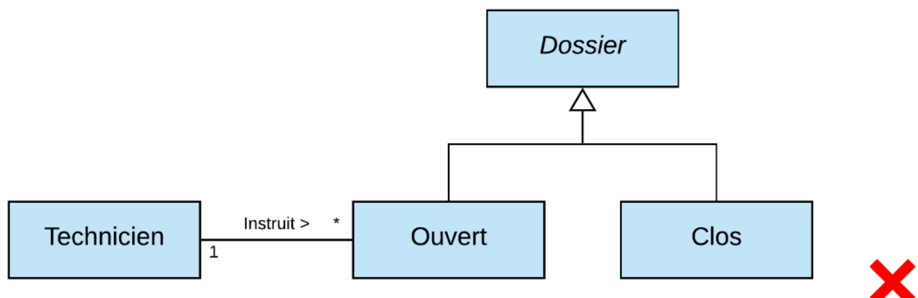
2.4 Rôle ou état ?

Nous allons adresser ici une erreur parfois commise (bien que moins fréquente que la confusion entre rôle et héritage). Il s'agit de la confusion dans un énoncé entre ce qui relève de l'héritage et de l'état.

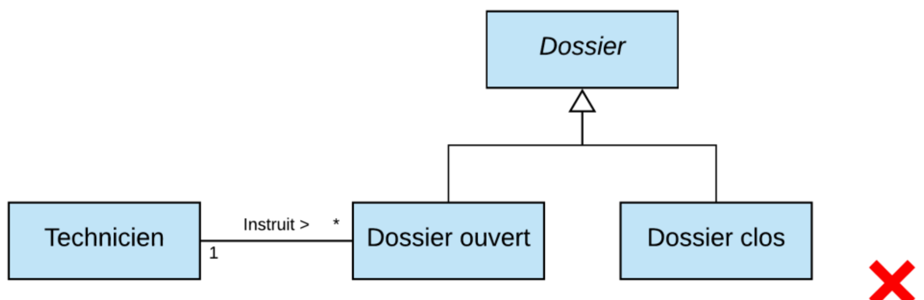
Considérons l'énoncé suivant :

> *Un technicien peut instruire des dossiers. Le dossier peut être « ouvert » ou « clos ». Le technicien ne peut instruire qu'un dossier « ouvert ».*

En première intention, nous aurions pu faire le diagramme suivant :

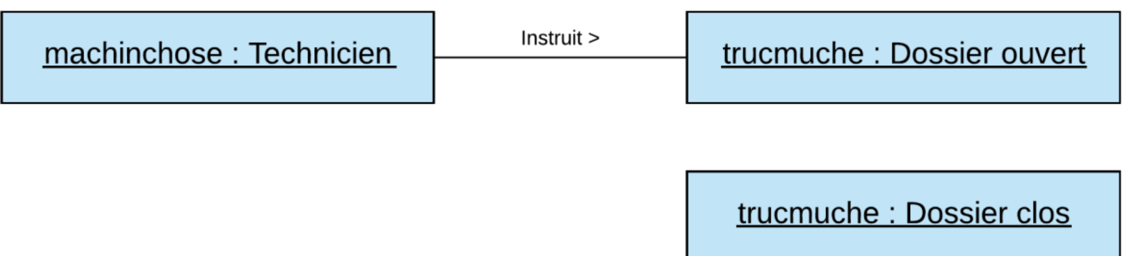


Ce diagramme est doublement faux. Tout d'abord, les noms des classes filles de dossier ne sont pas « autoporteurs ». C'est-à-dire qu'on ne peut pas les comprendre en dehors de lors contexte. D'une manière générale, quand une classe a pour nom un adjectif (ou ici un participe qui joue le rôle de), ce n'est pas bon signe. Corrigeons ce premier point avec d'attaquer le sujet majeur de ce paragraphe :



Nommer les deux sous-classes en « dossier ouvert » plutôt que « ouvert » (respectivement « dossier clos » et « clos »). Est bien plus correct.

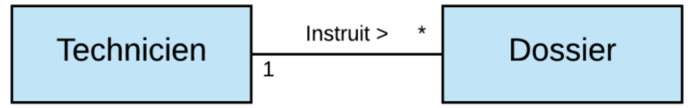
Mais le problème principal, on peut le découvrir en faisant tourner le modèle. Imaginez qu'un technicien « machinchose » traite le dossier « trucmuche ». Il le traite jusqu'à le clôturer. Si on en fait le diagramme objet, voici ce qu'on obtient :



Et là, rien ne va :

- L'objet du monde réel va se traduire par deux objets du modèle, en fonction du temps.
- Même si on vous écrit qu'un technicien ne peut instruire qu'un dossier ouvert, on peut avoir envie de savoir quel dossier il a traité même une fois traité. Ce modèle ne le permet pas.

En fait, le piège était de chercher à représenter les différents états du dossier dans un diagramme de classe. Ce n'est pas son rôle. Dans l'énoncé, la précision sur l'état du dossier devait être tout simplement ignoré pour la construction du diagramme de classe. Le modèle correct est le suivant :

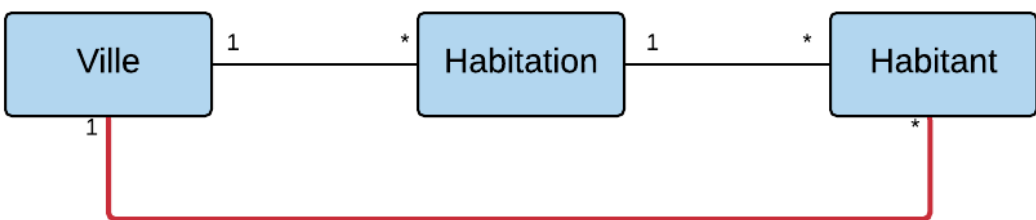


Remarque : nous pouvons à la rigueur ajouter un attribut « statut » à la classe dossier.

2.5 La transitivité

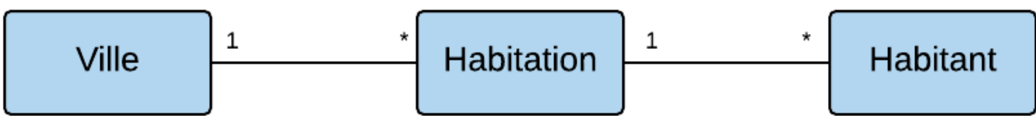
Il est parfois possible de ne pas représenter une relation entre deux classes lorsque celle-ci peut se déduire. C'est notamment le cas lorsqu'il est possible d'établir un lien en passant par un chemin de relations qui vont de l'une à l'autre. On parle dans ce cas de « relation par transitivité ».

Pour clarifier, prenons l'exemple suivant :



Bien que ce diagramme soit tout à fait valide, le lien ici en rouge est superflu. En effet, l'habitant a une seule habitation, qui se situe dans une seule ville. On peut donc savoir dans quelle ville vit un habitant en suivant les deux liens en noir. Ajouter le lien de ville vers habitant alourdit le schéma sans apporter d'information nouvelle.

Il faut donc préférer le diagramme suivant :



A noter que la transitivité est possible :

- Lorsque les associations sont « navigables » (pas de flèche) ;
- Lorsque les multiplicités le permettent. Retenez que c'est possible lorsque nous sommes sur du 0..1 ou 1 d'un côté et sur du * ou 1..* de l'autre sur chacun des de lien, et que l'étoile est toujours du même côté.

3.1 Granularité

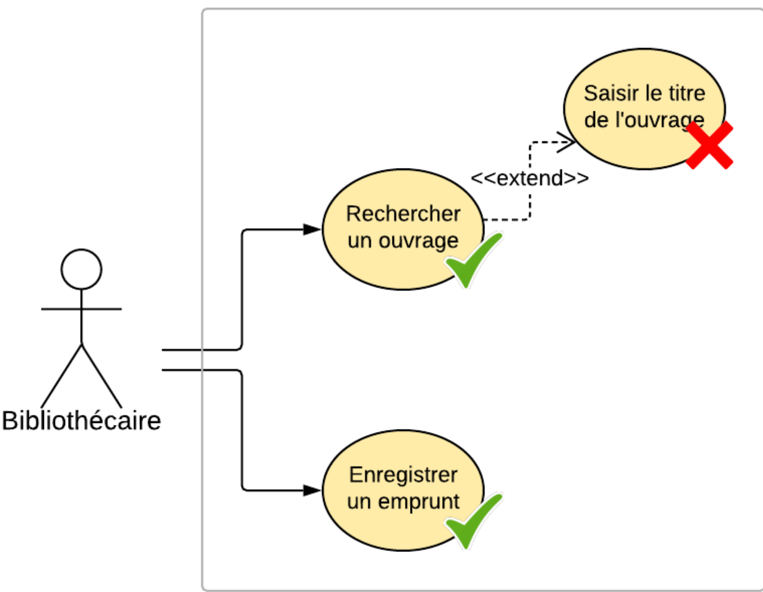
Le Langage UML ne dit rien de la granularité que doivent avoir les cas d'utilisation. Ce qui va guider ce type de choix est plutôt la méthode utilisée dans l'entreprise.

Le choix retenu lors de nos travaux pratique est de prendre l'orientation « cas d'utilisation métier » du TOGAF. Les caractéristiques sont :

- Un cas d'utilisation est une unité d'intention complète. Sa réalisation satisfait un besoin « métier », c'est-à-dire qu'il apporte une réponse à une problématique que se pose l'acteur (et non celle du système). C'est pour cette raison, par exemple, que nous excluons les cas d'utilisation « s'authentifier ». Ce n'est jamais une finalité. Quand on s'authentifie, c'est pour répondre à une contrainte du système pour pouvoir faire quelque chose d'autre.
- Il y a interaction entre l'acteur externe au système et le système lui-même. S'il n'y a pas interaction, c'est qu'il n'y pas besoin métier.

Un bon moyen de vérifier si on est sur la bonne granularité est d'utiliser le « test du patron ». Il s'agit, face à un cas d'utilisation, de se demander « est-ce qu'un patron pourrait me demander un truc pareil ».

Prenons l'exemple des cas d'un système de gestion de bibliothèque :



On peut imaginer que le responsable de la bibliothèque puisse demander à un de ses employés d'enregistrer un emprunt ou d'aller chercher un ouvrage. Il ne donnera en revanche jamais pour instruction de « saisir un titre ». La saisie du titre du logiciel n'est qu'une des actions que le bibliothécaire peut avoir à mener pour répondre à l'enjeu métier qu'est de « trouver un ouvrage », résultat attendu du cas d'utilisation « rechercher un ouvrage ».

3.2 Include, Extend

3.2.1 Représentation du lien

Avant de chercher à déterminer quand utiliser les liens « include » et « extend », il peut être utile de rappeler commet ils sont représentés graphiquement, et à quoi ils répondent :

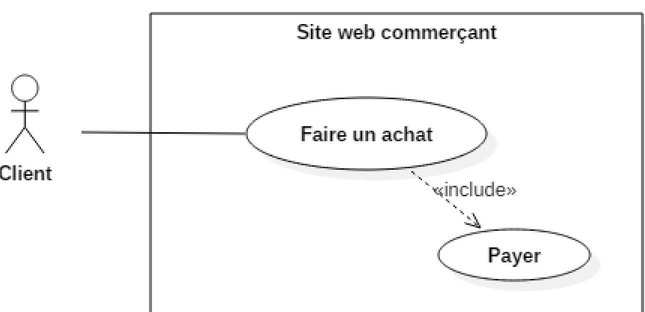
A «include» B	L'inclusion correspond au fait qu'un cas d'utilisation A fait obligatoirement appel au cas d'utilisation B dans son déroulé.	La flèche pointe vers le cas d'utilisation qui est inclut.
A B «extend»	L'extension signifie qu'un cas d'utilisation A peut éventuellement faire appel au d'utilisation B dans son déroulé.	La flèche pointe vers le cas d'utilisation qui peut inclure l'autre.

3.2.2 Quand faut-il lier les cas d'utilisations entre eux ?

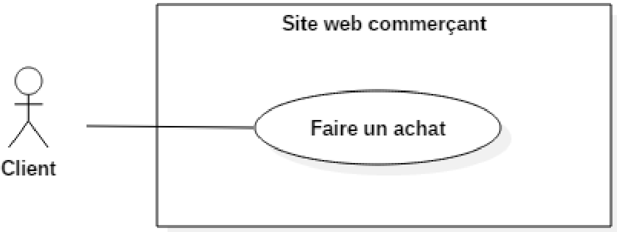
3.2.2.1 Le piège de la décomposition fonctionnelle

« Quand représenter ce lien ? » est une bonne question. Malheureusement, dans la pratique, la question qui est réellement posée est plutôt du type « dois-je ajouter un cas d'utilisation pour représenter cette fonctionnalité qui participe à la réalisation d'un use case ? ».

Par exemple, vous modélisez un site internet commerçant. Vous identifiez qu'un des cas d'utilisation est « faire un achat ». Vous identifiez par ailleurs que dans le cadre d'un achat, il y a une interaction qui consiste à payer. Vous en arrivez au diagramme suivant :



Le diagramme ci-dessus est valide d'un point de vue UML. Mais il n'est pas correct vis-à-vis de la philosophie de ce qu'est un cas d'utilisation. « Payer » n'est qu'une étape de faire un achat. Autrement dit, si vous n'aviez pas « faire un achat », jamais vous n'auriez identifié un cas « payer ». Pour cet exemple, le bon diagramme était tout simplement :



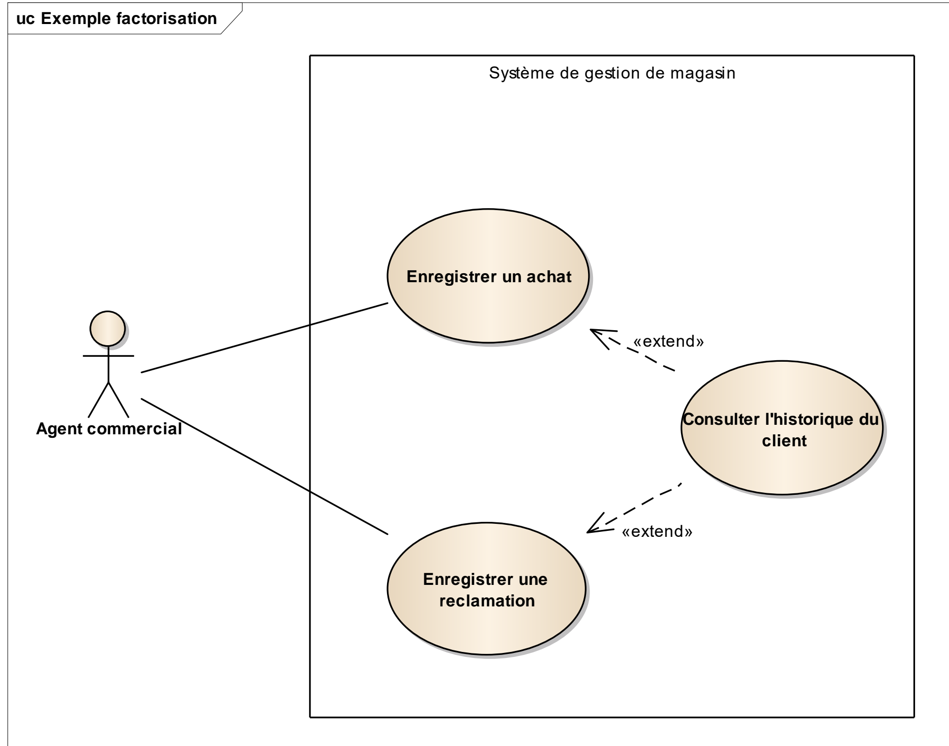
L'exercice 1 du TD 2 donne plusieurs exemples de ce qui ne se fait pas.

3.2.2.2 La factorisation

Il existe une situation où l'include trouve tout son intérêt : lorsque de cas d'utilisation distincts se partagent une grande partie de leur comportement.

Prenons par exemple la modélisation d'un système de gestion des achats de magasin. Lors de l'étude initiale nous identifions qu'il y (entre autres) les cas d'utilisation « enregistrer un achat » et « enregistrer une réclamation ».

Lors de la déclinaison des enchainements, nous nous rendons compte que dans un cas comme dans l'autre il est possible (sans être obligatoire) de « décrocher » vers la consultation de l'historique du client (dans un cas pour voir s'il est opportun de lui faire une ristourne, dans le second pour voir le contexte global du client). Il est alors possible de créer un nouveau cas « consulter l'historique du client » en support des deux autres cas, comme le montre le diagramme ci-dessous :

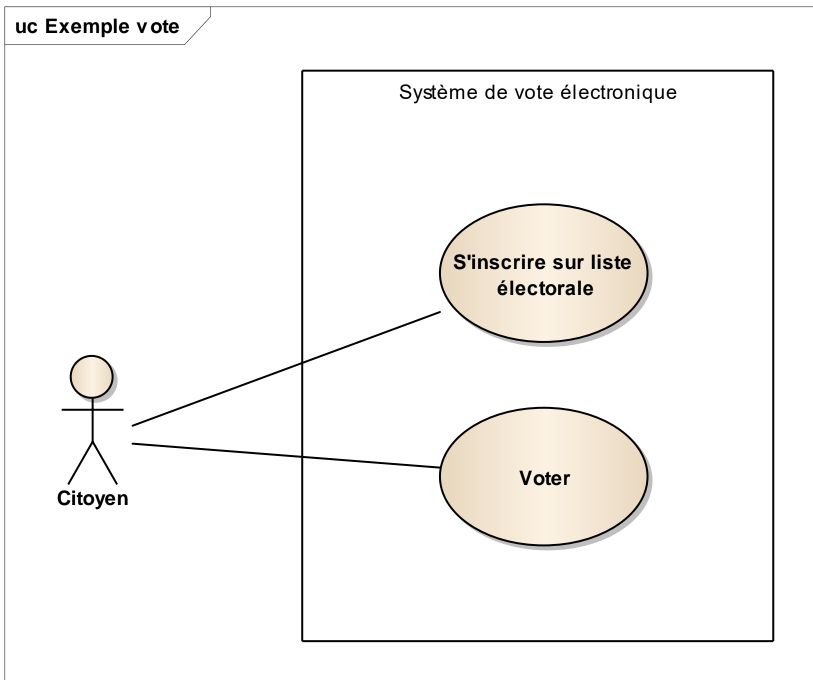


Attention tout de même de factoriser des séquences ayant une certaine taille. Cela n'a par exemple aucun sens de factoriser l'authentification à un système : c'est une action bien trop fine !

3.2.2.3 La temporalité

L'inclusion et l'extension signifie qu'un use case peut ou va participer à la réalisation d'un autre. Il ne peut donc exister un tel lien que si le déroulement des deux cas d'utilisation se fait dans le même temps.

Par exemple, si on considère ce que pourrait être un système de vote électronique :



Il pourrait être tentant de créer un lien entre les deux cas d'utilisation, du fait qu'il est nécessaire de s'inscrire pour voter. Cela se traduirait par un lien de type « extend » allant de « s'inscrire » vers « voter ».

Or, l'inscription se fait nécessairement bien avant. **Il n'y a donc aucun lien possible.**

L'inscription est en fait une « précondition » au vote. Ce type de contrainte n'est pas porté par le diagramme de cas d'utilisation. D'une manière générale, c'est plutôt porté par la description des enchainements.

C'est une erreur extrêmement courante, alors qu'elle est facile à éviter. Prenez bien le temps de vérifier ce point.

3.3 Proposition de démarche

On peut multiplier les exemples de ce qu'il faut faire et ne faut pas faire. Mais un bon moyen d'éviter de se tromper est d'adopter la démarche suivante :

- Chercher à identifier tous les cas d'utilisation du système en regardant les interactions entre les acteurs externes au système et le système lui-même, sans lier les cas d'utilisation les uns aux autres ;
- Vérifier qu'il n'existe pas de cas d'utilisation « trop fin », par exemple en utilisant le « test du patron » ;
- S'assurer que nous n'avons pas oublié les cas d'utilisation « cœur de système » (par exemple, le cas d'utilisation « ouvrir porte » de l'exercice 4 du TD 2) ;
- S'il existe des liens d'inclusion ou d'extension dans le diagramme, les ajouter ;
- Au moment de produire la description des cas (que ce soit l'enchainement ou les diagrammes de séquence), voire s'il n'y a pas des fonctionnements communs entre plusieurs cas d'utilisation, que vous pouvez alors factoriser.

Appliquer cette démarche va souvent vous conduire à des diagrammes de cas d'utilisation sans « include » ni « extend ». Cela ne doit pas vous déranger : c'est tout à fait normal !

4.1 Ce qu'il faut savoir sur les transitions

4.1.1 Syntaxe

Un point important sur le diagramme d'états-transitions qui peut paraitre basique mais qu'il faut pourtant maitriser, c'est la syntaxe :

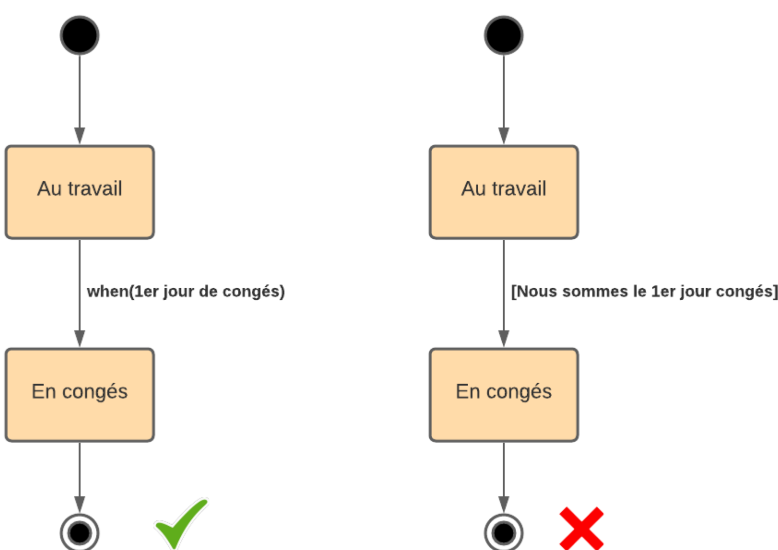
A noter :

- Si la transition ne comporte ni évènement, ni condition de garde alors on passe immédiatement de l'état 1 à l'état 2.

La description de l'évènement est libre. Il existe cependant deux mots clés bien utiles :

- When(condition) : on attend que la condition soit vraie pour activer la transition ;
- After(durée) : on attend une certaine durée pour activer la transition.

D'une manière générale, il y a peu de chances de se tromper. Il y a tout de même une erreur à ne pas commettre :

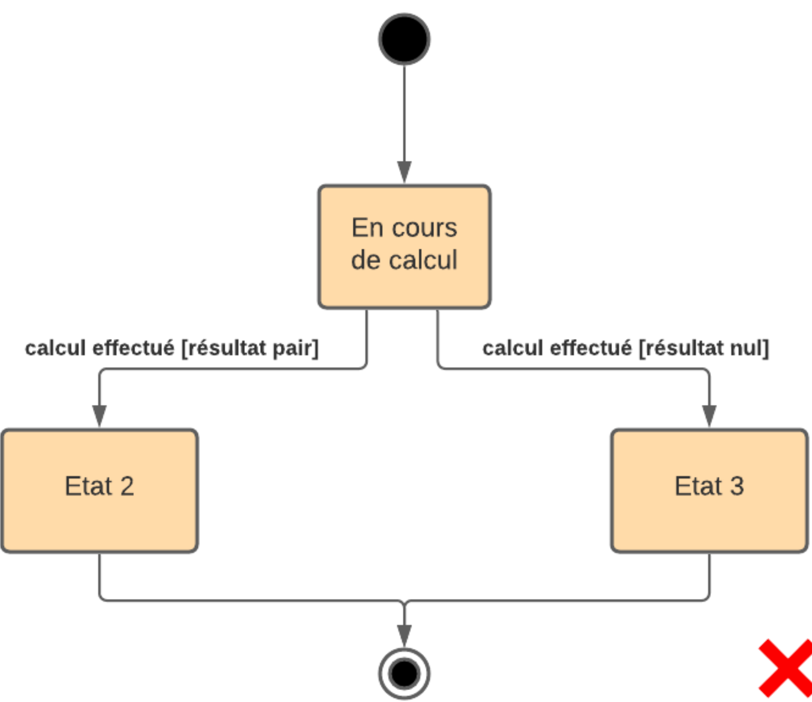


Dans le diagramme de droite, comme il n'y a aucun évènement, une fois que nous sommes sur l'état « au travail » on évalue tout de suite la condition de garde. Comme elle est fausse, on ne passe pas à

l'état « en congés ». Mais comme il n'y a aucune notion d'attente, cette condition n'est jamais réévaluée. En fait, dans le diagramme de droite, nous resterons coincés *ad vitam aeternam* à l'état « au travail ».

4.1.2 Il faut pouvoir sélectionner une transition sortante

Le principe est simple : quand on quitte un état, on doit toujours savoir quelle flèche suivre. Regardons le contre-exemple suivant :



Ce modèle est faux pour deux raisons :

- Les transitions sortantes doivent être mutuellement exclusives. Or, si le résultat est « zéro », comme zéro est un nombre pair, il est impossible de décider s'il faut aller sur l'état 2 et l'état 3 ;
- Il faut qu'il y ait au moins une transition activable. Or, si le résultat est « trois », nous sommes coincés.

4.2 Plusieurs états simultanés ?

La plupart des diagrammes d'états-transitions partent du principe qu'à un instant T, nous sommes dans un unique état. Pourtant, cela n'est pas toujours suffisant :

- Nous pouvons avoir à gérer des états totalement indépendants ;
- Durant le cycle de vie, il est possible qu'il y ait un « tronçon » ou – temporairement – notre système peut prendre deux états simultanés.

Nous allons voir ici comment représenter ces situations.

4.2.1 Etats totalement indépendants

Prenons le cas de la classe « salle de cours ». On considère que :

- La salle peut être allumée ou éteinte ;
- Qu'elle peut être vide ou occupée.

Bien évidemment, les deux situations sont totalement indépendantes : il peut y avoir de la lumière que la salle soit vide ou occupée, par exemple.

Il y a deux façons de traiter la chose

4.2.1.1 Deux diagrammes

La première, la plus simple, est de considérer que notre classe possède deux états : l'état d'éclairage et l'état d'occupation. Chacun de ces états est indépendant et possède son propre diagramme.

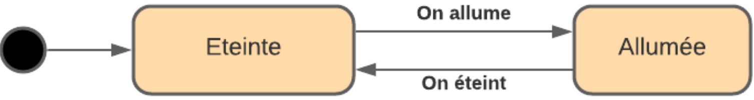


Diagramme d'état pour l'état d'éclairage

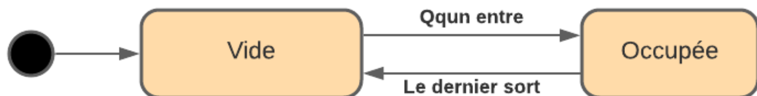
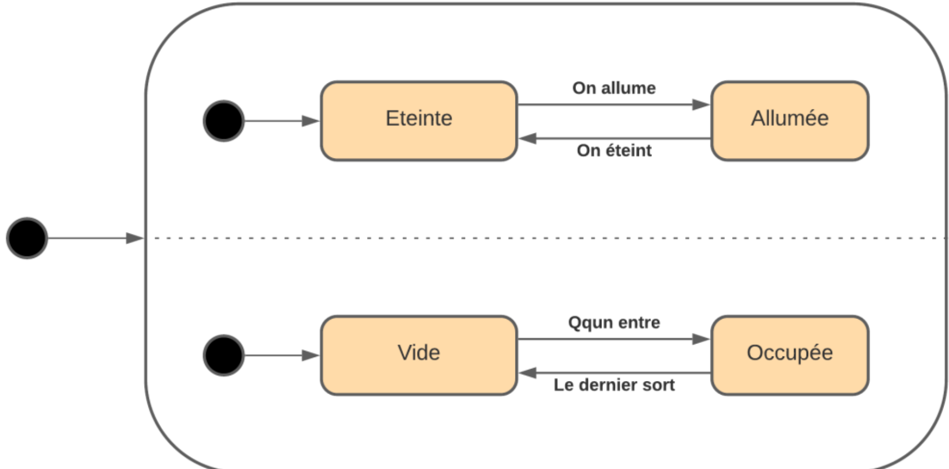


Diagramme d'état pour l'état d'occupation

A noter que ce n'est jamais la solution attendue dans le cadre des travaux dirigés ou même de l'examen final.

4.2.1.2 Un seul diagramme, avec état composite

L'autre façon de faire est de représenter un état composite.



On peut noter au passage que nous n'avons pas d'état de fin. On considère qu'il n'est pas utile d'évoquer la fin de ce cycle de vie qui serait… eh bien… la destruction de la salle ?

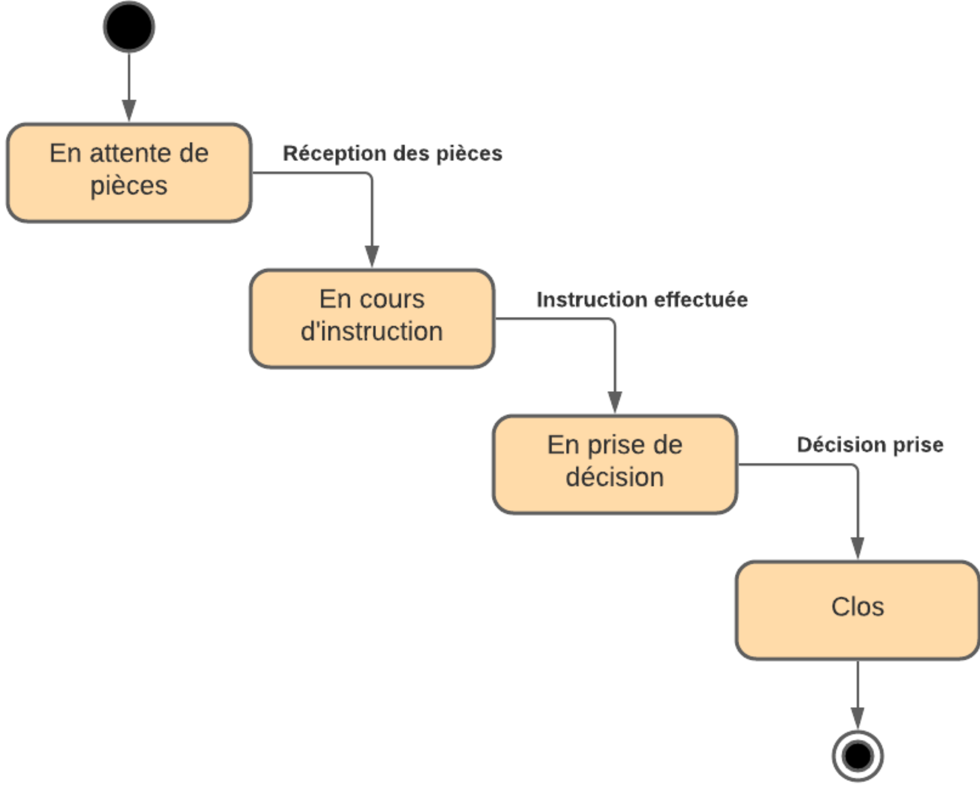
4.2.2 Etats temporairement indépendants

Imaginons l'énoncé suivant :

Représentez le diagramme d'état de la classe dossier d'instruction dont le processus est le suivant :

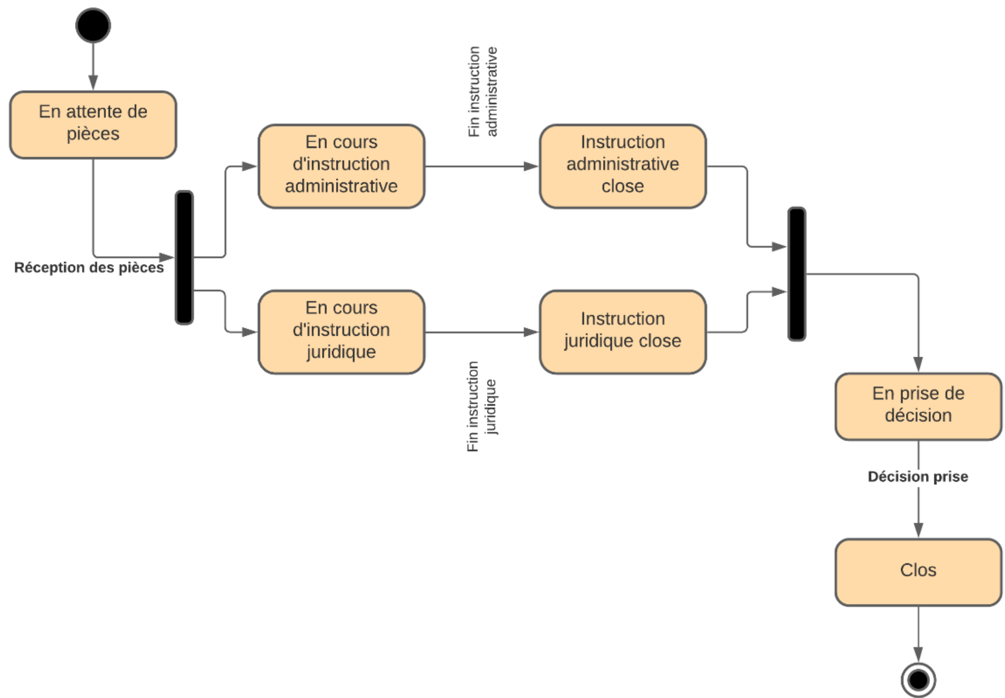
Le dossier est créé dès l'envoi du courrier de demandes de pièces. Une fois les pièces reçues, elles sont envoyées pour instruction à la fois au service administratif et au service juridique. Une fois que les deux services ont répondu, on entre en prise de décision. Le dossier est clos une fois la décision prise.

En première intention, nous aurions pu faire le diagramme suivant :

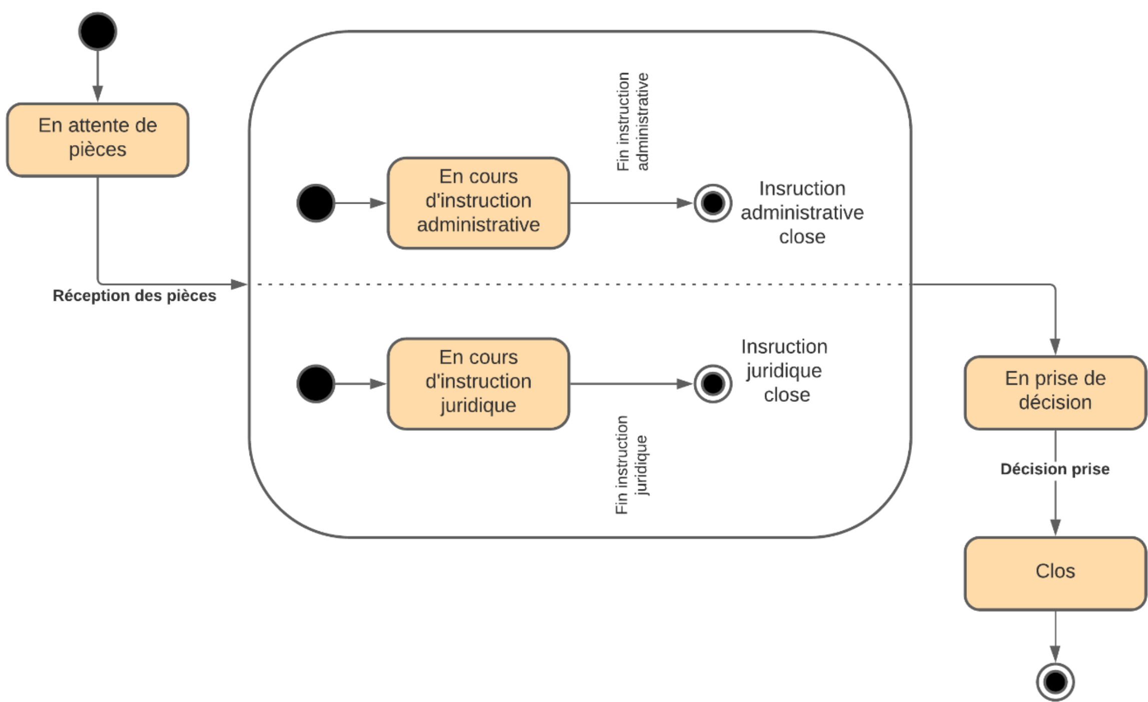


Sauf que, nous l'avons vu, l'instruction se passe à deux endroits en parallèle. Nous aimerions donc pouvoir rendre compte de cela en présentant où nous en sommes de chaque côté. Il existe plusieurs façons de représenter cela. Elles sont toutes en commun de mettre en œuvre des fork/join ou des états complexes (voire les deux).

Si on choisit de représenter pour chaque service un état « en cours d'instruction » et « un état terminé », cela pourrait donner les diagrammes présentés ci-dessous.



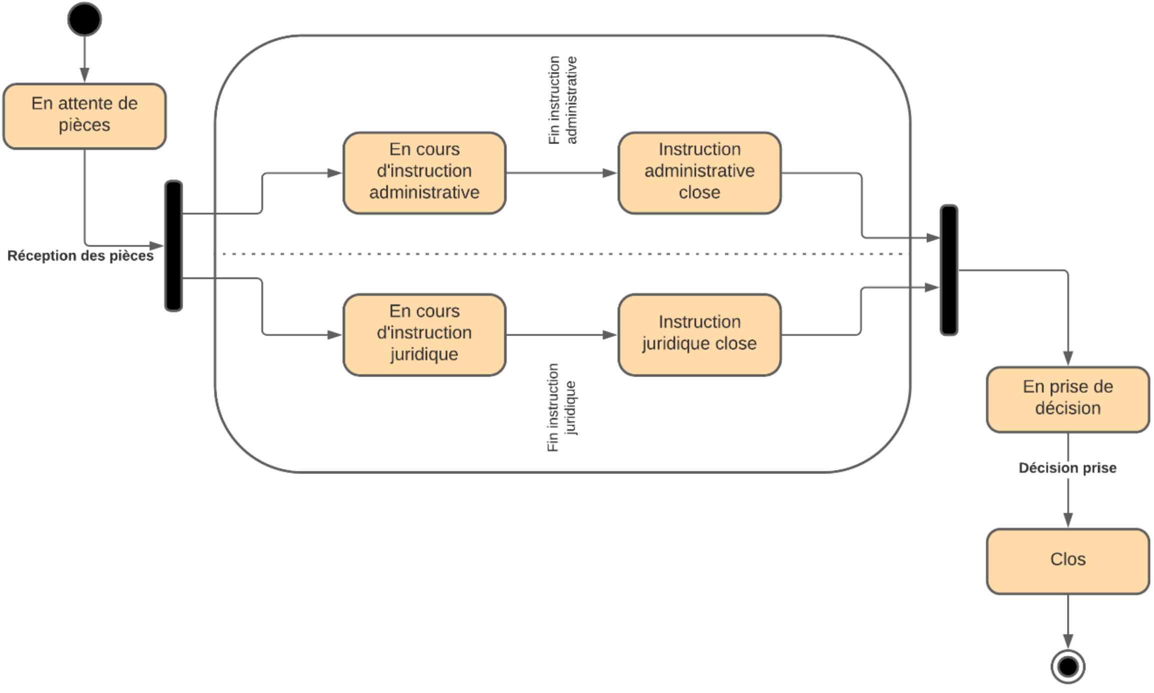
Avec fork/join, mais sans super-état. Lorsque les deux instructions seront réalisées, les deux transitions en sortie seront activées, ce qui permettra



Ici, dès que les deux diagrammes d'états inclus dans l'état composite arriveront à la fin, alors nous passerons à la prise de décision. A noter ici qu'il est préférable de nommer les pseudo-états de fin

internes : à un instant T cela permettra de dire « l'instruction administrative est en cours, l'instruction juridique est close ».

Et enfin :



Ce schéma est certes un peu complexe, mais présente l'avantage d'être le plus explicite.

4.3 Considérations pratique

Une fois le modèle construit :

- Vérifiez que l'ensemble des transitions est correctement décrit ;
- Pour chaque état, vérifiez que toutes les conditions d'activation des transitions sortantes sont mutuellement exclusives ;
- Pour chaque état, vérifiez qu'il y a toujours au moins un chemin possible pour aller sur un autre état ;
- Vérifiez que vous avec bien au moins un événement de début ;
- Vérifiez que tous les chemins mènent jusqu'à un évènement de fin (préférablement unique).

5.1 Modélisation de la recherche

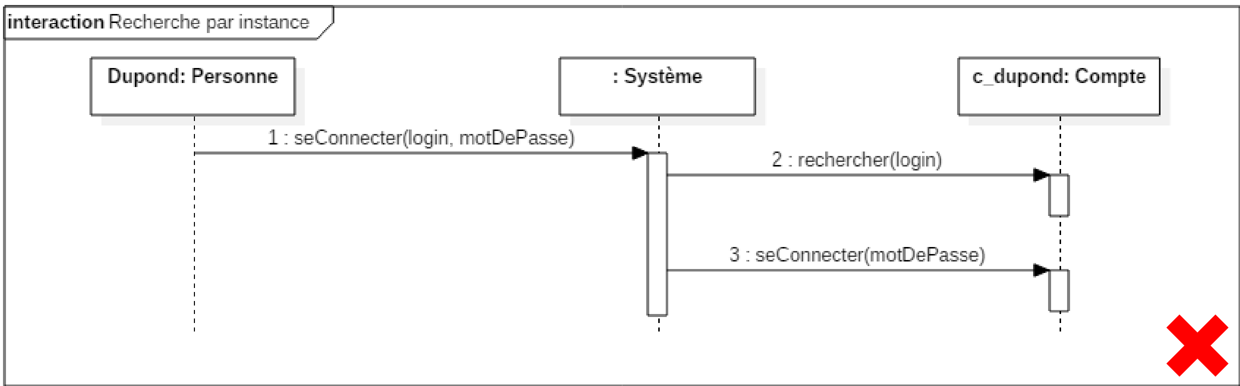
5.1.1 Problématique

Dans l'exercice 2 du TD 3, nous avions à modéliser la recherche d'un compte dont nous connaissions le login. Dans l'exercice 3 du TD 3, nous avions la même problématique avec un vélo.

Dans les deux cas, nous sommes sur un niveau proche de l'implémentation. Au niveau du diagramme de séquence - contrairement à un niveau « système », où on ne représente que le système dans sa globalité et les acteurs qui gravitent autour - nous avons à représenter l'ensemble des objets mis en œuvre.

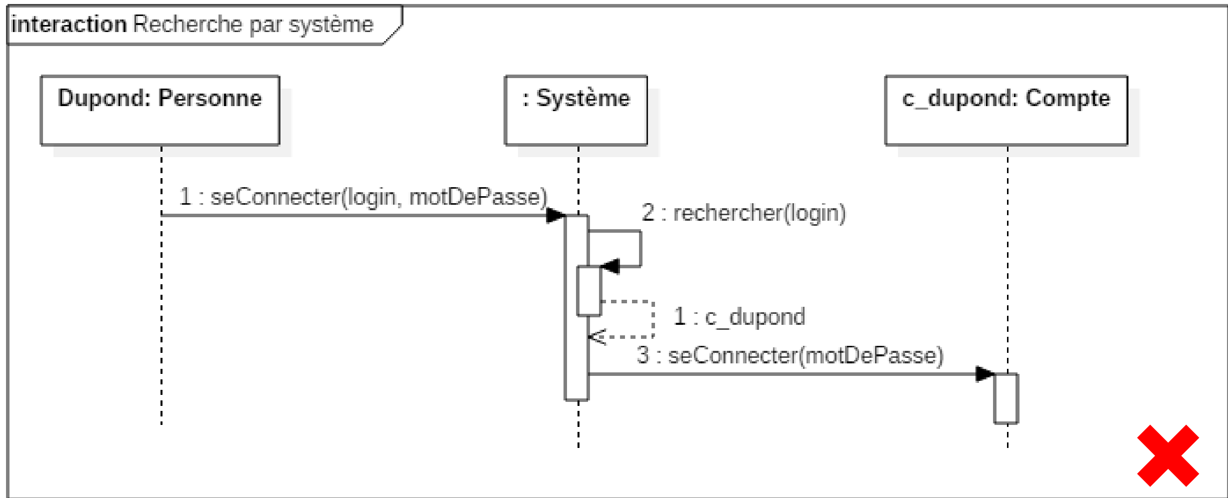
5.1.2 Ce qu'il ne faut pas faire

En premier réflexe, vous pourriez être tentés de faire quelque chose de cette forme :



On part ici du principe que nous allons avoir la ligne de vie du compte de monsieur Dupond (l'instance c_dupond) avec laquelle nous allons interagir. Ce qui est effectivement le cas. Sauf qu'il n'est pas possible de l'utiliser pour effectuer la recherche, vu que c_dupond est le résultat de la recherche. On ne peut pas utiliser cette instance avant de l'avoir trouvée !

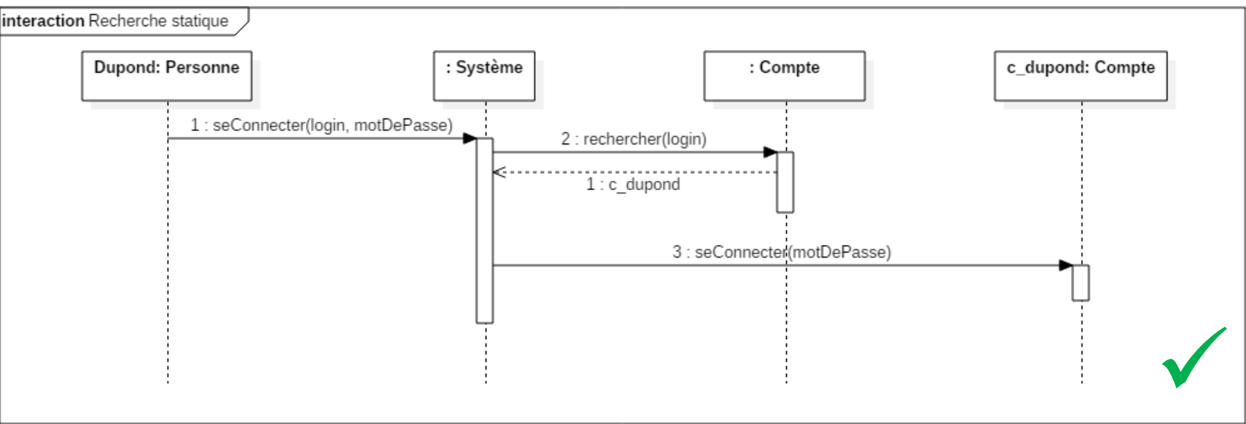
L'autre erreur à ne pas commettre est d'utiliser le système pour effectuer la recherche :



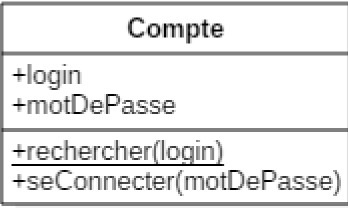
La raison est un peu plus subtile. Il s'agit ici d'une erreur de conception plus que de logique. En conception objet, on regroupe les données et les traitements dans des ensembles qu'on veut cohérents (les classes). A contrario, le système n'a pas à connaitre le détail du fonctionnement de tous les objets avec lesquels il interagit. En d'autres termes, il n'a pas à implémenter les différentes mécaniques de recherche.

5.1.3 Pattern 1 : la recherche portée par la classe de l'objet cherché

C'est la proposition vue en TD. Il s'agit aussi de la manière la plus simple de gérer la problématique, même si elle requiert d'utiliser une opération de recherche sur le compte :



La particularité de ce diagramme est que la recherche n'est pas invoquée sur l'instance c_dupond (que l'on cherche), mais sur un objet non nommé. En fait, la recherche est indépendante de toute instance de « Compte ». Elle est peut (et donc elle doit) être portée directement par la classe. Cela se matérialiste par le fait que l'opération doit être définie comme étant statique :

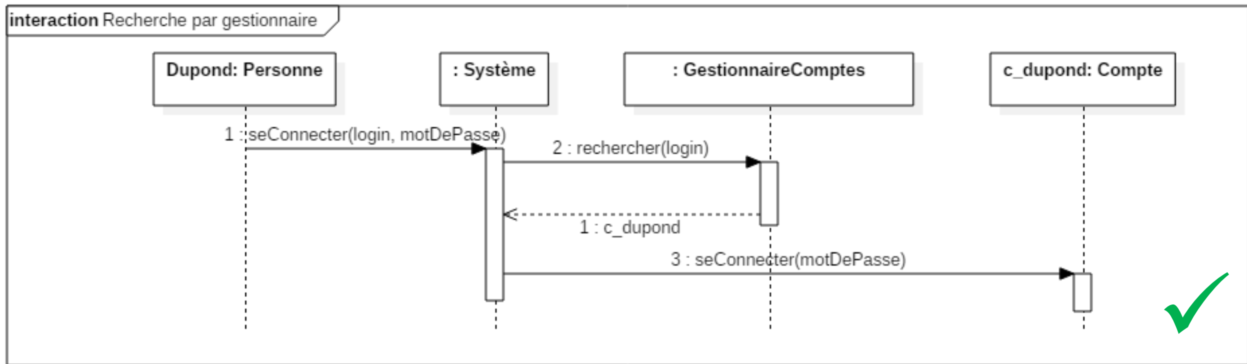


En UML, la notation pour une opération statique consiste à la souligner. A noter qu'on peut de la même façon définir des attributs statiques.

5.1.4 Pattern 2 : la recherche portée par une classe « gestionnaire d'ensemble »

Dans ce pattern, nous introduisions une troisième classe entre le système et la classe dont on veut créer un objet. Appliqué à l'exercice 2, cela pourrait être une classe « gestionnaire de comptes ». Dans l'exercice 3, cela pourrait être une classe « flotte de vélos ». Pour faire court, l'idée est ici d'avoir une classe modélisant non pas un exemplaire de la classe, mais l'ensemble.

Dans ce cas de figure, le diagramme de séquence serait de la forme :



Nous voyons ici apparaître le gestionnaire de comptes qui joue son rôle d'intermédiaire lors de la recherche.

5.1.5 Considérations pratiques

On parle ici de « pattern » parce qu'il s'agit d'un motif de conception réutilisable. L'idée est que face à une situation telle qu'évoqué lors des exemples, il suffit d'appliquer la solution proposée ici sans trop se poser de question.