ÉTUDE D'UN SYSTÈME DE RÉSERVATION DE VOL

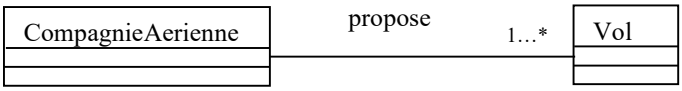
Cette étude de cas concerne un système simplifié de réservation de vols pour une agence de voyages.
Les interviews des experts métier auxquelles on a procédé ont permis de résumer leur connaissance du domaine sous la forme des phrases suivantes :

1. Des compagnies aériennes proposent différents vols.
2. Un vol est ouvert à la réservation et refermé sur ordre de la compagnie.
3. Un client peut réserver un ou plusieurs vols, pour des passagers différents.
4. Une réservation concerne un seul vol et un seul passager.
5. Une réservation peut être annulée ou confirmée.
6. Un vol a un aéroport de départ et un aéroport d'arrivée.
7. Un vol a un jour et une heure de départ, et un jour et une heure d'arrivée.
8. Un vol peut comporter des escales dans des aéroports.
9. Une escale a une heure d'arrivée et une heure de départ.
10. Chaque aéroport dessert une ou plusieurs villes.

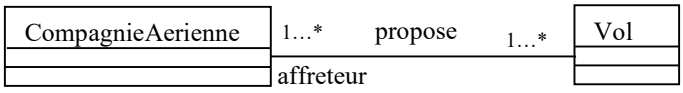
Nous allons entreprendre progressivement la réalisation d'un modèle statique d'analyse (aussi appelé modèle du domaine) à partir de ces « morceaux de connaissance ».

Étape 1– Modélisation des phrases 1 et 2

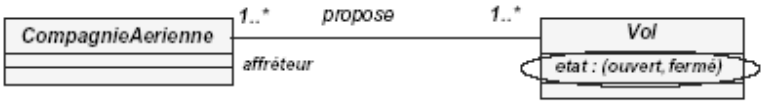
En premier lieu, nous allons modéliser la première phrase :
1. Des compagnies aériennes proposent différents vols.
CompagnieAerienne et *Vol* sont des concepts importants du monde réel ; ils ont des propriétés et des comportements. Ce sont donc des classes candidates pour notre modélisation statique.
Nous pouvons initier le diagramme de classes comme suit :



La multiplicité « 1..* » du côté de la classe *Vol* a été préférée à une multiplicité « 0..* » car nous ne gérons que les compagnies aériennes qui proposent au moins un vol.
En revanche, la phrase ne nous donne pas d'indication sur la multiplicité du côté de la classe *CompagnieAerienne* . C'est là une première question qu'il faudra poser à l'expert métier.
Par la suite, nous partirons du principe qu'un vol est proposé le plus souvent par une seule compagnie aérienne, mais qu'il peut également être partagé entre plusieurs affréteurs.
Au passage, on notera que le terme « affréteur » est un bon candidat pour nommer le rôle joué par la classe *CompagnieAérienne* dans l'association avec *Vol*



Nous allons maintenant nous intéresser à la deuxième phrase. Les notions d'ouverture et de fermeture de la réservation représentent des concepts dynamiques. Il s'agit en effet de changements d'état d'un objet *Vol* sur ordre d'un objet *CompagnieAérienne* . Une solution naturelle consiste donc à introduire un attribut énuméré *etat* , comme cela est montré sur la figure suivante.



En fait, ce n'est pas vraiment la bonne approche : tout objet possède un état courant, en plus des valeurs de ses attributs. Cela fait partie des propriétés intrinsèques des concepts objets. La notion d'état ne doit donc pas apparaître directement en tant qu'attribut sur les diagrammes de classes : elle sera modélisée dans le point de vue dynamique grâce au diagramme d'états . Dans le diagramme de classes UML, les seuls concepts dynamiques disponibles sont les opérations. Or, les débutants en modélisation objet ont souvent du mal à placer les opérations dans les bonnes classes ! Plus généralement, l'assignation correcte des responsabilités aux bonnes classes est une qualité distinctive des concepteurs objets expérimentés…

<u>Question 1</u> : Dans quelle classe placez-vous les opérations que l'on vient d'identifier (ouverture et fermeture de la réservation**)?**

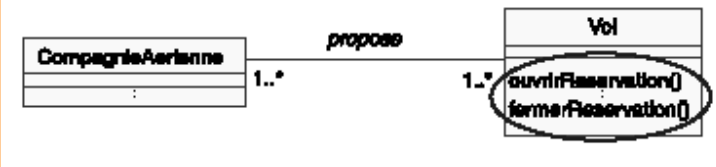
<u>Indication :</u>

Qui est ouvert à la réservation ?

Solution :

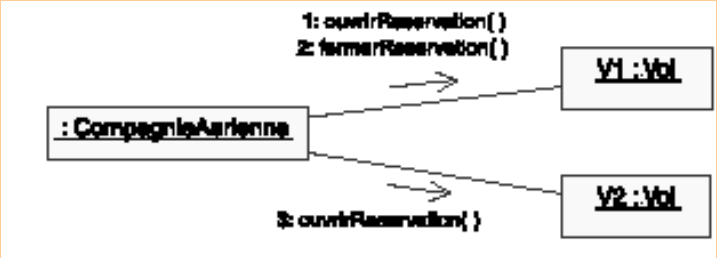
C'est le vol, et non pas la compagnie.

En orienté objet, on considère que l'objet sur lequel on pourra réaliser un traitement doit le déclarer en tant qu'opération. Les autres objets qui posséderont une référence dessus pourront alors lui envoyer un message qui invoque cette opération. Il faut donc placer les opérations dans la classe *Vol* , et s'assurer que la classe *CompagnieAérienne* a bien une association avec elle.



L'association *propose* s'instanciera en un ensemble de liens entre des objets des classes *CompagnieAerienne* et *Vol* .

Elle permettra donc bien à des messages d'ouverture et de fermeture de réservation de circuler entre ces objets, comme cela est indiqué sur le diagramme de collaboration ci-après.

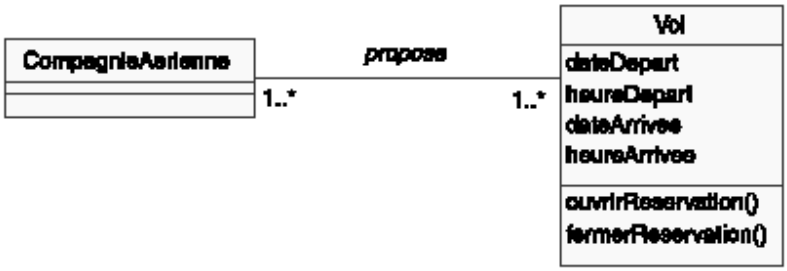


Étape 2 – Modélisation des phrases 6, 7 et 10

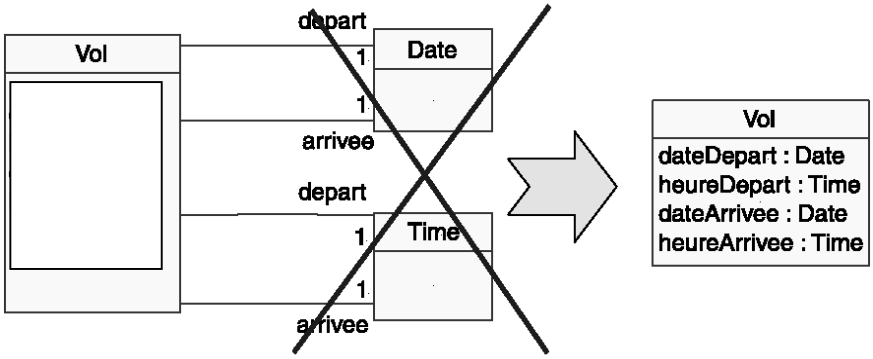
Poursuivons la modélisation de la classe *Vol* . Les phrases 6 et 7 s'y rapportent directement. Considérons tout d'abord la phrase 7 :

7. Un vol a un jour et une heure de départ et un jour et une heure d'arrivée.

Toutes ces notions de dates et d'heures représentent simplement des valeurs pures. Nous les modéliserons donc comme des attributs et pas comme des objets à part entière.
Un objet est un élément plus « important » qu'un attribut. Un bon critère à appliquer en la matière peut s'énoncer de la façon suivante : si l'on ne peut demander à un élément que sa valeur, il s'agit d'un simple attribut ; si plusieurs questions s'y appliquent, il s'agit plutôt d'un objet qui possède lui-même plusieurs attributs, ainsi que des liens avec d'autres objets.

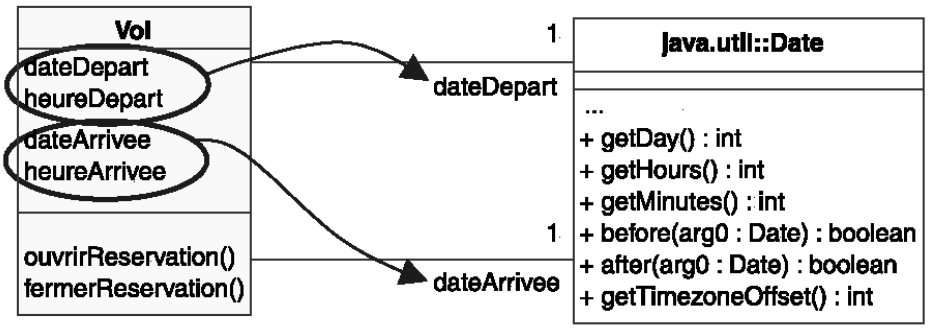


NOTA
DATE COMME TYPE NON PRIMITIF
Nous avons expliqué précédemment pourquoi nous préférons modéliser les dates et heures comme des attributs et pas des objets, contrairement aux aéroports.
Une solution plus sophistiquée a été proposée par M. Fowler a : elle consiste à créer une classe Date et à s'en servir ensuite pour typer l'attribut au lieu d'ajouter une association.



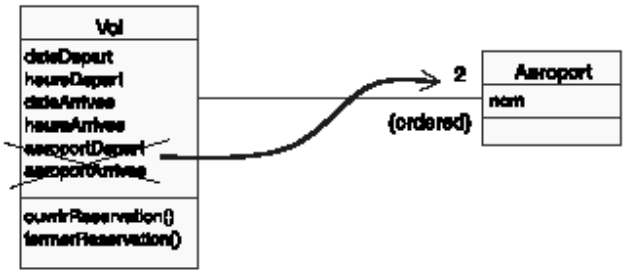
DIFFÉRENCE ENTRE MODÈLE D'ANALYSE ET DE CONCEPTION
Dans le code Java de l'application finale, il est certain que nous utiliserons explicitement la classe d'implémentation Date (du package java.util).

Il ne s'agit pas là d'une contradiction, mais d'une différence de niveau d'abstraction, qui donne lieu à deux modèles différents, un modèle d'analyse et un modèle de conception détaillée, pour des types de lecteurs et des objectifs distincts.

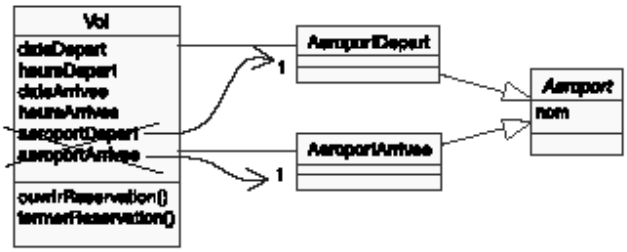


Question 2 : Essayer d'appliquer ce principe à la phase 6, je vous propose différente solution de modélisation discuter de leurs avantages et inconvénients

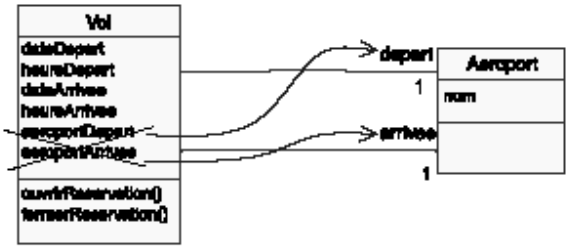
Sol :1



sol :2



sol :3



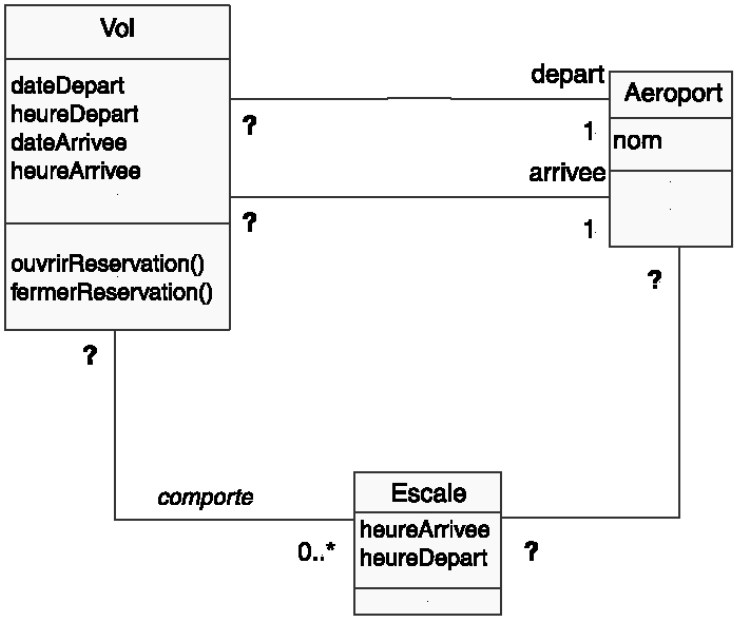
solution :
Contrairement aux notions d'heure et de date qui sont des types « simples », la notion d'aéroport est complexe ; elle fait partie du « métier ». Un aéroport ne possède pas seulement un nom, il a aussi une capacité, dessert des villes, etc. C'est la raison pour laquelle nous préférons créer une classe *Aeroport* plutôt que de simples attributs *aeroportDepart* et *aeroportArrivee* dans la classe *Vol*
.
Une première solution consiste à créer une association avec une multiplicité 2 du côté de la classe *Aeroport* . Mais nous perdons les notions de départ et d'arrivée.

Une astuce serait alors d'ajouter une contrainte { ordered } du côté *Aeroport* , pour indiquer que les deux aéroports liés au vol sont ordonnés (l'arrivée a toujours lieu après le départ !) cf sol1.
Il s'agit là d'une modélisation « tordue » que nous ne recommandons pas, car elle n'est pas très parlante pour l'expert métier et il existe une bien meilleure solution… Une autre solution tentante consiste à créer deux sous-classes de la classe *Aeroport cf sol2*

Pourtant, cette solution est incorrecte ! En effet, tout aéroport est successivement aéroport de départ pour certains vols et aéroport d'arrivée pour d'autres. Les classes *AeroportDepart* et *AeroportArrivee* ont donc exactement les mêmes instances redondantes, ce qui devrait décourager d'en faire deux classes distinctes.
Le concept UML de rôle s'applique parfaitement dans cette situation. La façon la plus précise de procéder consiste donc à créer deux associations entre les classes *Vol* et *Aeroport* , chacune affectée d'un rôle différent avec une multiplicité égale à 1 exactement.

Question 3 **Étape 3 – Modélisation des phrases 8 et 9**
Considérons maintenant les escales, c'est-à-dire les phrases 8 et 9.
8. Un vol peut comporter des escales dans des aéroports.
9. Une escale a une heure d'arrivée et une heure de départ.
Chaque escale a deux propriétés d'après la phrase 9 : heure d'arrivée et heure de départ. Elle est également en relation avec des vols et des aéroports, qui sont eux-mêmes des objets, d'après la phrase 8. Il est donc naturel d'en faire une classe à son tour. Cependant, la phrase 8 est aussi imprécise : une escale peut-elle appartenir à plusieurs vols, et quelles sont les multiplicités entre *Escale* et *Aeroport* ? De plus, le schéma n'indique toujours pas les multiplicités du côté *Vol* avec *Aeroport*.

Complétez les multiplicités des associations !!!

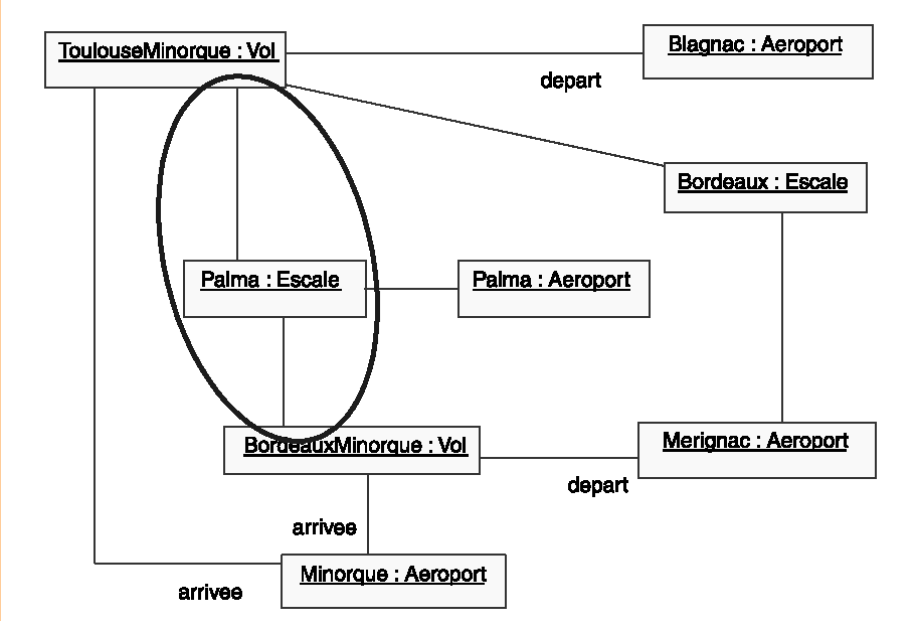


solution

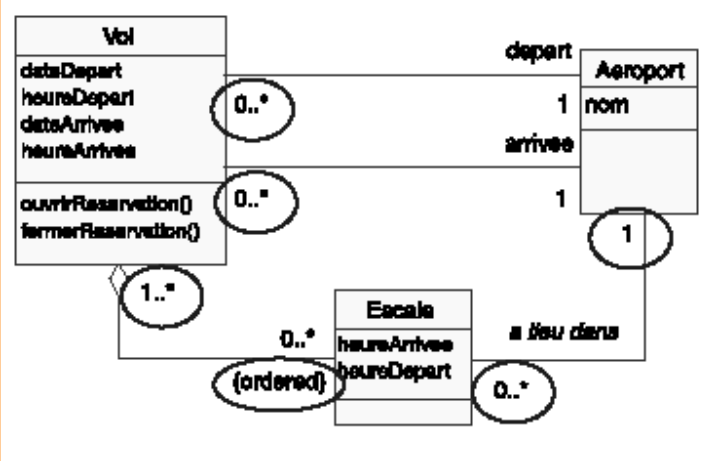
D'après la phrase 8, un vol peut comporter des escales dans des aéroports. Cette formulation est ambiguë, et vaut que l'on y réfléchisse un peu, en faisant même appel aux conseils d'un expert métier.

On peut commencer par ajouter les multiplicités entre *Escale* et *Aeroport*, ce qui doit être aisé. Il est clair qu'une escale a lieu dans un et un seul aéroport, et qu'un aéroport peut servir à plusieurs escales. De même, un aéroport peut servir de départ ou d'arrivée à plusieurs vols.

On pourrait également penser qu'une escale n'appartient qu'à un vol et un seul, mais est-ce bien certain ? Après consultation de l'expert métier, un contre-exemple nous est donné, sous forme du diagramme d'objets suivant.



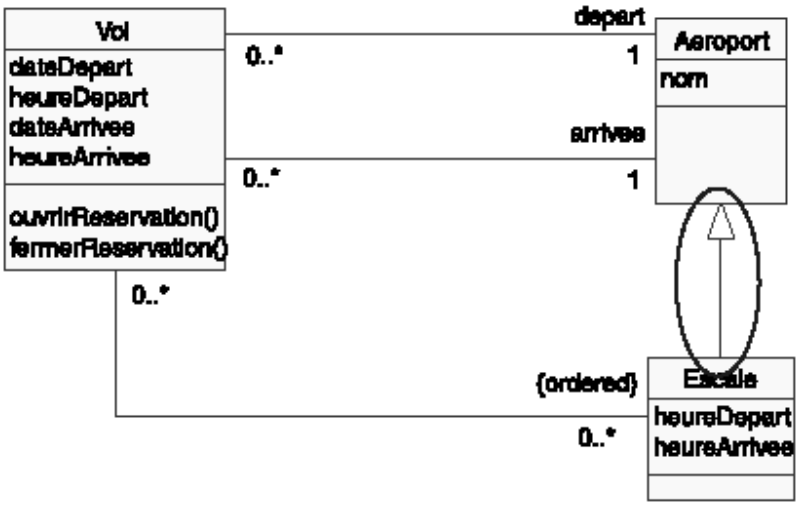
Une escale peut donc appartenir à deux vols différents, en particulier quand ces vols sont « imbriqués ». Notez combien il est efficace de recourir au diagramme d'objets pour donner un exemple, ou encore un contre-exemple, qui permette d'affiner un aspect délicat d'un diagramme de classes.
Pour finaliser le diagramme des phrases 8 et 9, il nous suffit d'ajouter deux précisions :
• l'association entre *Vol* et *Escale* est une agrégation (mais certainement pas une composition, puisqu'elle est partageable) ;
• les escales sont ordonnées par rapport au vol.



Question 5 : **Classe d'association**

Dans la solution précédente, la classe Escale sert d'intermédiaire entre les classes *Vol* et *Aeroport.* Elle a peu de réalité par elle-même, et cela nous laisse donc penser à une autre solution la concernant...

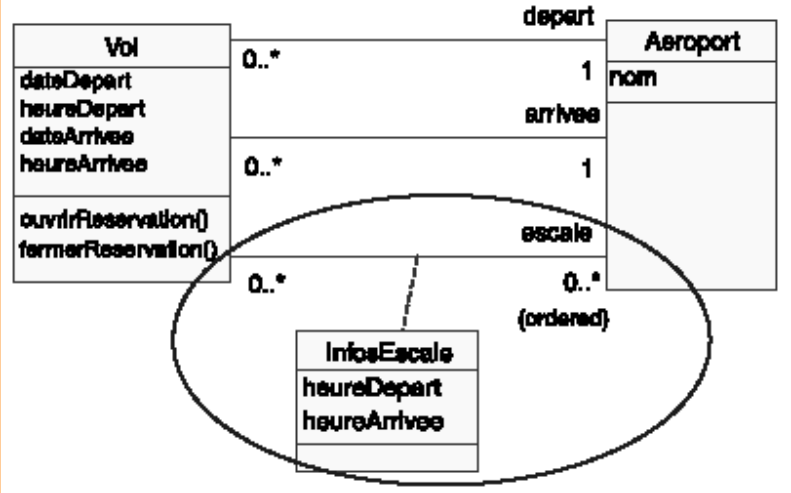
Proposez une solution plus sophistiquée pour la modélisation des escales.
Elément de réponse
Au vu du diagramme précédent, il apparaît que la classe *Escale* comporte peu d'informations propres ; elle est fortement associée à *Aeroport* (multiplicité 1) et n'existe pas par elle-même, mais seulement en tant que partie d'un *Vol*. Une première idée consiste à considérer *Escale* comme une spécialisation de *Aeroport*.



Il s'agit là d'une solution séduisante, l'escale récupérant automatiquement par héritage le nom d'aéroport, et ajoutant par spécialisation les heures de départ et d'arrivée. Pourtant, on ne peut pas recommander d'y recourir. En effet, peut-on vraiment dire qu'une escale est une sorte d'aéroport, peut-on garantir que la classe *Escale* est conforme à 100 % aux spécifications de sa super-classe ? Est-ce qu'une escale dessert des villes, est-ce qu'une escale peut servir de départ ou d'arrivée à un vol ? Si l'on ajoute des opérations *ouvrir* et *fermer* à la classe *Aéroport*, s'appliqueront-elles à *Escale* ? Il ne s'agit donc pas, à vrai dire, d'un héritage d'interface, mais bien plutôt d'une facilité dont pourrait user un concepteur peu scrupuleux pour récupérer automatiquement l'attribut *nom* de la classe *Aeroport*, avec ses futures méthodes d'accès. Cette utilisation de l'héritage est appelée un héritage d'implémentation et elle est de plus en plus largement déconseillée. En outre, si l'on veut un jour spécialiser au sens métier les aéroports en aéroports internationaux et régionaux, par exemple, on devra immédiatement gérer un héritage multiple.

Quelle est donc votre solution ?

Solution :
Pourquoi ne pas considérer plutôt cette notion d'escale comme un troisième rôle joué par un aéroport par rapport à un vol ? Les attributs *heureArrivee* et *heureDepart* deviennent alors des attributs d'association, comme cela est montré sur le schéma suivant. La classe *Escale* disparaît alors en tant que telle, et se trouve remplacée par une classe d'association *InfosEscale*. On notera l'homogénéité des multiplicités du côté de la classe *Vol*.



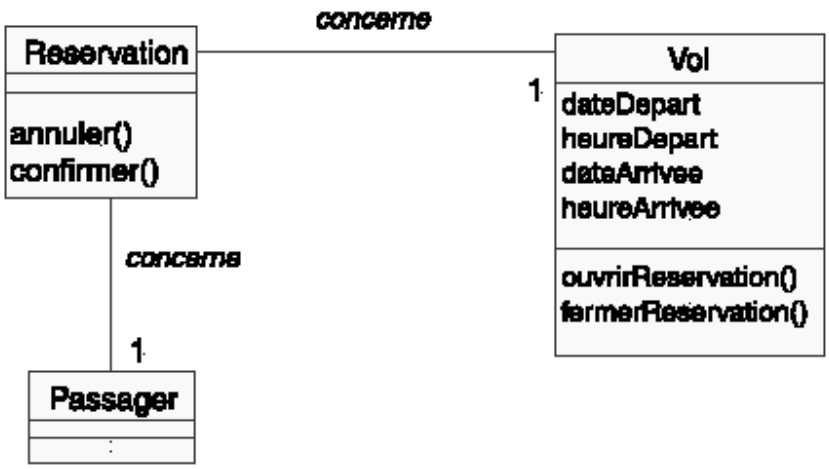
Étape 4 – Modélisation des phrases 3, 4 et 5

Nous pouvons maintenant aborder le traitement du concept de réservation.
Relisons bien les phrases 3 à 5 qui s'y rapportent directement.
3. Un client peut réserver un ou plusieurs vols, pour des passagers différents.
4. Une réservation concerne un seul vol et un seul passager.
5. Une réservation peut être annulée ou confirmée.
Une question préliminaire peut se poser immédiatement…

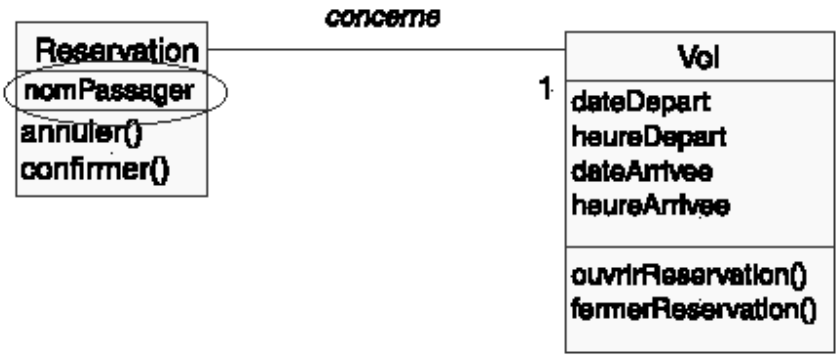
Faut-il vraiment distinguer les concepts de client et de passager ?

Discutons !

Au premier abord, cette distinction peut paraître superflue, mais elle est en fait absolument nécessaire ! Prenons le cas des déplacements professionnels : le client est souvent l'employeur de la personne qui se déplace pour son travail. Cette personne joue alors le rôle du passager et apprécie de ne pas devoir avancer le montant de son billet. Le concept de client est fondamental pour les aspects facturation et comptabilité, alors que le concept de passager est plus utile pour les aspects liés au vol lui même (embarquement, etc.).
D'après la phrase 4, une réservation concerne un seul vol et un seul passager. Nous pouvons modéliser cela directement par deux associations.
La phrase 5, quant à elle, se traduit simplement par l'ajout de deux opérations dans la classe *Reservation*, sur le modèle de la phrase 2.



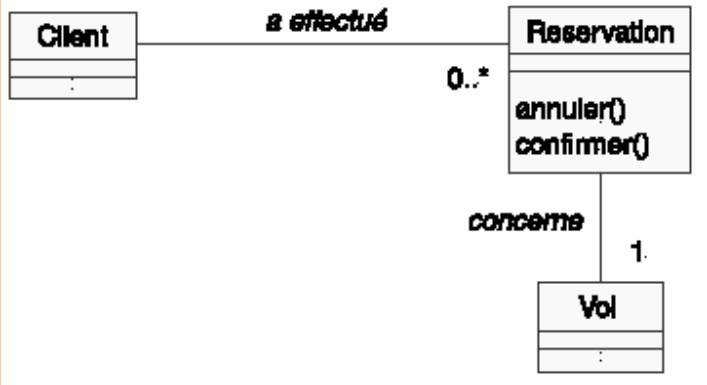
Notez néanmoins qu'une solution plus concise est possible, à savoir considérer le passager comme un simple attribut de *Reservation*. L'inconvénient majeur concerne la gestion des informations sur les passagers. En effet, il est fort probable que l'on ait besoin de gérer les coordonnées du passager (adresse, téléphone, e-mail, etc.), voire des points de fidélité, ce que ne permet pas facilement la solution simpliste montrée sur la figure suivante.



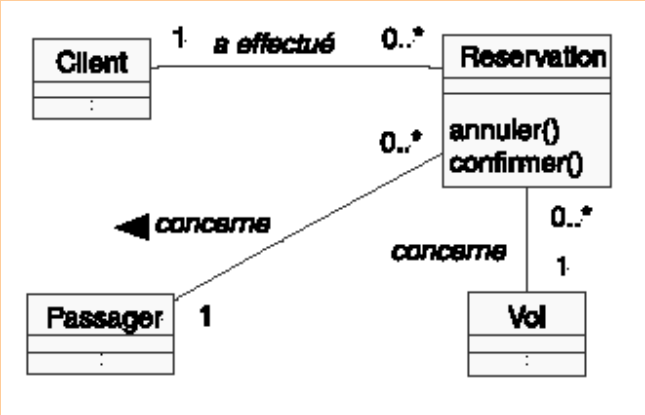
Nous conserverons donc l'approche faisant de *Passager* une classe à part entière.

EXERCICE 7.
Modélisation de la réservation
Poursuivons notre traitement du concept de réservation. La phrase 3 est un peu plus délicate, à cause de sa formulation alambiquée.
Modélisez la phrase 3 et complétez les multiplicités des associations précédentes.

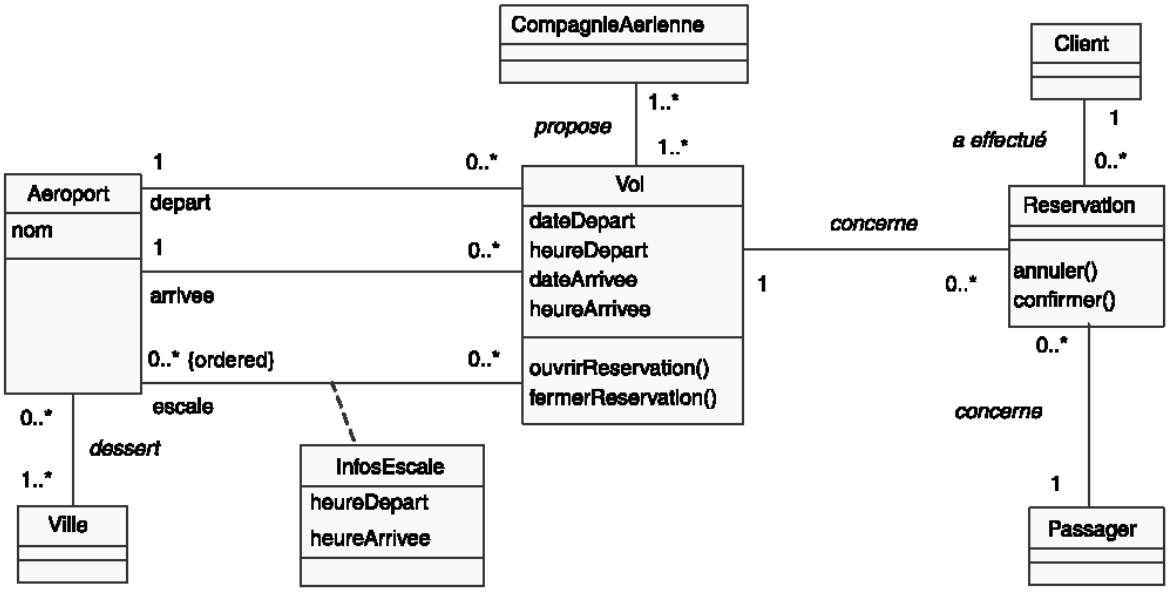
Solution :
Le début de la phrase 3 peut prêter à confusion en raison de la relation directe qu'il semble impliquer entre client et vol. En fait, le verbe « réserver » masque le concept déjà identifié de réservation. Nous avons vu lors de la modélisation de la phrase 4 qu'une réservation concerne un seul vol. Le début de la phrase 3 peut donc se reformuler plus simplement : un client peut effectuer plusieurs réservations (chacune portant sur un vol). Ce qui se traduit directement par le schéma suivant.



Nous allons compléter le diagramme en ajoutant d'abord les deux multiplicités manquantes. Il est clair qu'une réservation a été effectuée par un et un seul client, et qu'un même vol peut être concerné par zéro ou plusieurs réservations. Ajoutons ensuite la classe *Passager* et complétons les multiplicités. Combien de réservations un même passager peut-il avoir ? Au premier abord, au moins une, sinon il n'est pas passager. En fait, nous gérons des réservations, pas le vol lui-même. Nous avons donc besoin de stocker des instances persistantes de passagers, même s'ils n'ont pas tous de réservation à l'instant courant. Encore une question de point de vue de modélisation ! Pour l'application qui gère l'embarquement, un passager a une et une seule réservation, mais ici il faut prévoir « 0..* ».



conclusion temporaire
Le modèle que l'on obtient par la modélisation des 10 phrases de l'énoncé ressemble actuellement au diagramme de la figure présentée ci-après.



Certaines classes n'ont pas d'attribut, ce qui est plutôt mauvais signe pour un modèle d'analyse représentant des concepts métier. La raison en est simplement que nous n'avons identifié que les attributs qui sont directement issus des phrases de l'énoncé. Il en manque donc certainement…
Pourquoi ?

EXERCICE 3-8.

Ajout d'attributs métier
Ajoutez les attributs métier qui vous semblent indispensables.

Solution

Pour chacune des classes, nous répertorions ci-après les attributs indispensables. Attention ! On ne doit pas lister dans les attributs des références à d'autres classes : c'est le but même de l'identi.cation des associations.

Aeroport :
- nom

Client :
- nom
- prenom
- adresse
- numTel
- numFax

CompagnieAerienne :
- nom

InfosEscale :
- heureDepart
- heureArrivee

Passager :
- nom
- prenom

Reservation :
- date
- numero

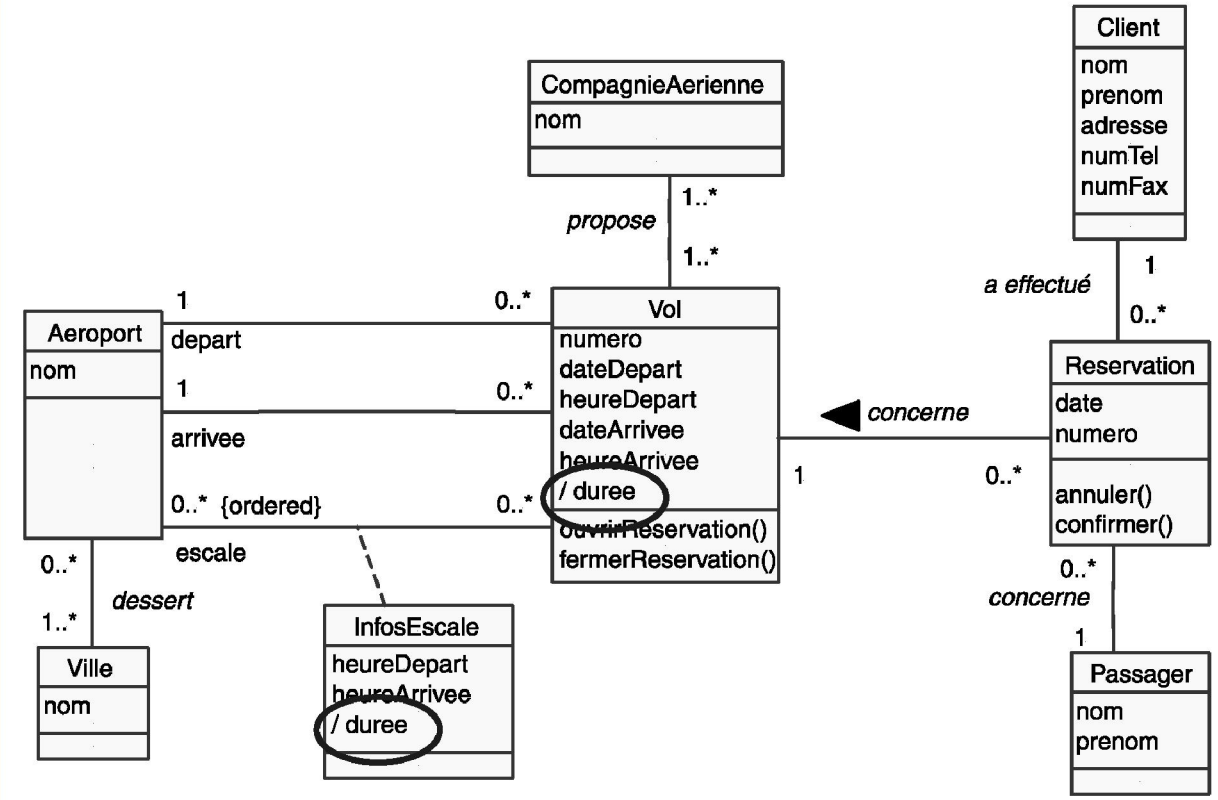
Ville :
- nom

Vol :
- numero
- dateDepart
- heureDepart
- dateArrivee
- heureArrivee

On notera que c'est la convention de nommage recommandée par les auteurs d'UML qui est utilisée ici.
À retenir CONVENTIONS DE NOMMAGE EN UML
Les noms des attributs commencent toujours par une minuscule (contrairement aux noms des classes qui commencent systématiquement par une majuscule) et peuvent contenir ensuite plusieurs mots concaténés, commençant par une majuscule.
Il est préférable de ne pas utiliser d'accents ni de caractères spéciaux. Les mêmes conventions s'appliquent au nommage des rôles des associations, ainsi qu'aux opérations.

EXERCICE 3-9. Ajout d'attributs dérivés
Complétez le modèle avec quelques attributs dérivés pertinents.

Solution :
Un attribut dérivé est une propriété valuée intéressante pour l'analyste, mais redondante, car sa valeur peut être déduite d'autres informations disponibles dans le modèle.
Un bon exemple en est fourni par la notion de *durée* d'un vol. En effet, il est clair que cette information est importante : le client voudra certainement connaître la durée de son vol, sans qu'il doive la calculer lui-même ! Le système informatique doit être capable de gérer cette notion. Or, les informations nécessaires pour cela sont déjà disponibles dans le modèle grâce aux attributs existants relatifs aux dates et heures de départ et d'arrivée. Il s'agit donc bien d'une information dérivée.
Le même raisonnement s'applique pour la durée de chaque escale.
Le diagramme présenté ci-après récapitule le nouvel état de notre modèle avec tous les attributs.



<u>à retenir :</u>

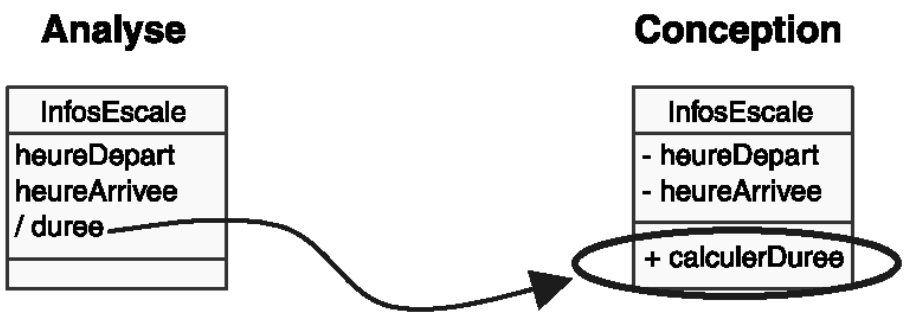
ATTRIBUT DÉRIVÉ EN CONCEPTION

Les attributs dérivés permettent à l'analyste de ne pas faire de choix de conception prématuré. Cependant, ce concept n'existant pas dans les langages objets, le concepteur va être amené à choisir entre plusieurs solutions :

– Garder un attribut « normal » en conception, qui aura ses méthodes d'accès (get et set) comme les autres attributs : il ne faut pas oublier de déclencher la mise à jour de l'attribut dès qu'une information dont il dépend est modifiée.

– Ne pas stocker la valeur redondante, mais la calculer à la demande au moyen d'une méthode publique.

La seconde solution est satisfaisante si la fréquence de requête est faible, et l'algorithme de calcul simple. La première approche est nécessaire si la valeur de l'attribut dérivé doit être disponible en permanence, ou si le calcul est très complexe et coûteux. Comme toujours, le choix du concepteur est une affaire de compromis…


Analyse
Conception
InfosEscale
heureDepart
heureArrivee
/ duree
InfosEscale
- heureDepart
- heureArrivee
+ calculerDuree