

## Analyse - Point de vue statique

### Chapitre 3

## Étude de cas : système de réservation de vols

---

#### Objectifs du chapitre

Ce chapitre va nous permettre d'illustrer, pas à pas, à partir d'une nouvelle étude de cas, les principales difficultés que pose la construction des diagrammes de classes UML.

Le diagramme de classes a toujours été le diagramme le plus important dans toutes les méthodes orientées objet. C'est celui que les outils de génération automatique de code utilisent en priorité. C'est également celui qui contient la plus grande gamme de notations et de variantes, d'où la difficulté d'utiliser correctement tous ces concepts.

---

#### Éléments mis en jeu

- Classe, objet
- Opération
- Association, multiplicité
- Attribut, attribut dérivé
- Agrégation, composition
- Classe d'association, qualificatif
- Contraintes, méta-classe
- Packages
- Généralisation, classe abstraite.

---

### ÉTUDE DE CAS 3 - ÉNONCÉ

---

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 conceptuel statique 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 :

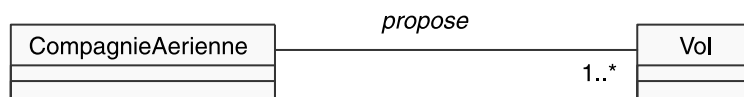


Figure 3-1. Modélisation de la phrase 1

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 *CompagnieAerienne* dans l'association avec *Vol*.

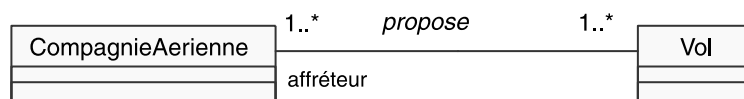


Figure 3-2. Modélisation complétée de la phrase 1

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 *CompagnieAerienne*. Une solution naturelle consiste donc à introduire un attribut énuméré *etat*, comme cela est montré sur la figure suivante.

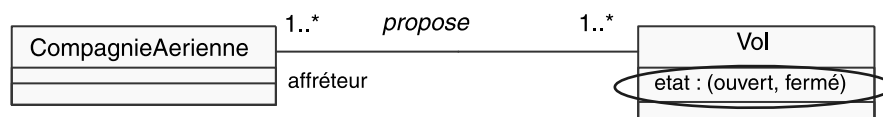


Figure 3-3. Première modélisation de la phrase 2

En fait, ce n'est pas 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 (voir chapitres 5 et 6). 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...



Question \*

3-1 DANS QUELLE CLASSE PLACEZ-VOUS LES OPÉRATIONS QUE L'ON VIENT D'IDENTIFIER ?



...

### RÉPONSE 3-1

Qui est ouvert à la réservation ? 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.

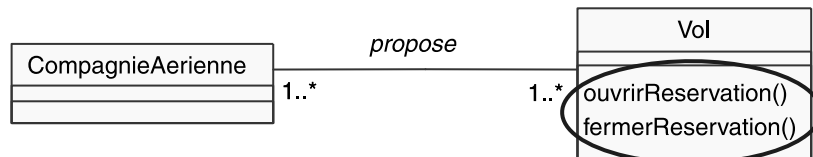


Figure 3-4. Modélisation correcte de la phrase 2

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.

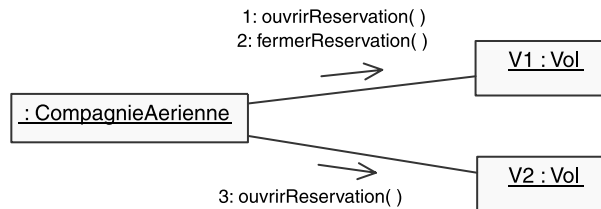


Figure 3-5. Diagramme de collaboration illustrant la phrase 2

## 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.

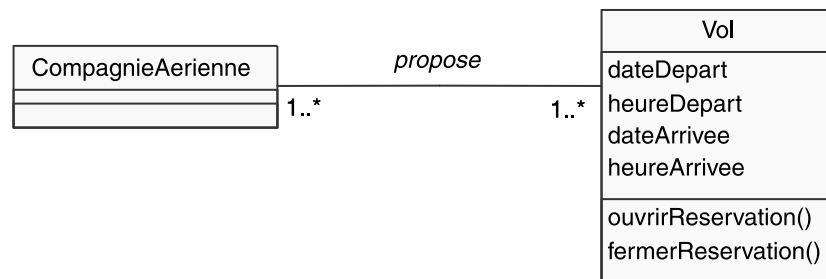


Figure 3-6. Modélisation des phrases 1, 2 et 7

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.

Essayez d’appliquer ce principe à la phrase 6 :

6. Un vol a un aéroport de départ et un aéroport d’arrivée.



Question\*\*\*

3-2 QUELLES SONT LES DIFFÉRENTES SOLUTIONS POUR MODÉLISER LA PHRASE 6, AVEC LEURS AVANTAGES ET INCONVÉNIENTS ?



...

### RÉPONSE 3-2

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 *Aéroport* plutôt que de simples attributs *aéroportDepart* et *aéroportArrivee* dans la classe *Vol*.

Une première solution consiste à créer une association avec une multiplicité 2 du côté de la classe *Aéroport*. Mais nous perdons les notions de départ et d'arrivée. Une astuce serait alors d'ajouter une contrainte *{ordered}* du côté *Aéroport*, pour indiquer que les deux aéroports liés au vol sont ordonnés (l'arrivée a toujours lieu après le départ !).

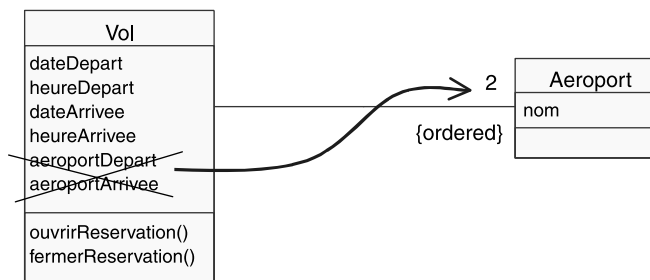


Figure 3-7. Solution approximative de la phrase 6

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 *Aéroport*.

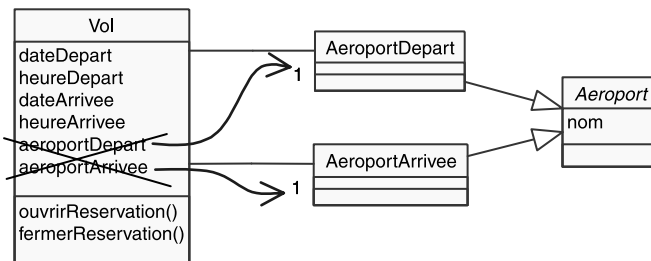


Figure 3-8. Solution incorrecte de la phrase 6

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.

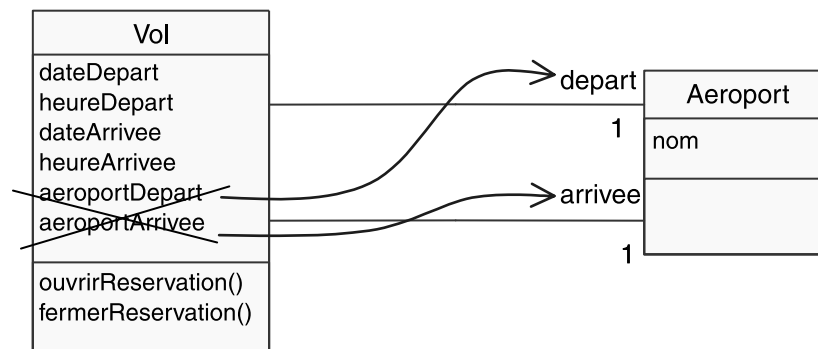


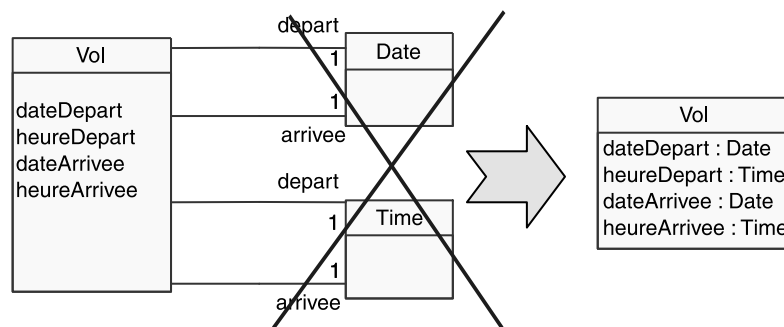
Figure 3-9. Modélisation correcte de la phrase 6



### 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<sup>a</sup> : elle consiste à créer une classe utilisateur Date et à s'en servir ensuite pour typer l'attribut au lieu d'ajouter une association.



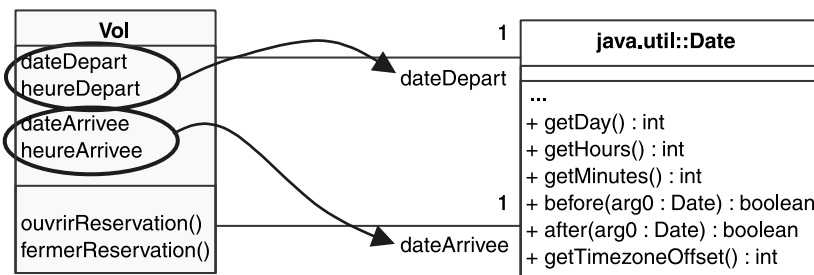
Modélisation de la classe utilisateur Date comme type non primitif

a. *Analysis Patterns: Reusable Object Models*, M. Fowler, 1997, Addison-Wesley.



### 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`).



Fragment possible de diagramme de classe de conception détaillée avec Java

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.



Nous pouvons maintenant passer à la modélisation de la phrase 10.

10. Chaque aéroport dessert une ou plusieurs villes.

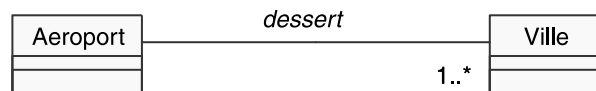


Figure 3-10. Modélisation de la phrase 10

Cependant, nous constatons une fois encore que la phrase 10 ne traduit qu'un seul sens de l'association. Elle ne permet pas de déterminer la multiplicité du côté de la classe *Aéroport*. La question doit donc être formulée de la façon suivante : par combien d'aéroports une ville est-elle desservie ?



Question \*

3-3 QUELLE EST LA MULTIPLICITÉ DU CÔTÉ AÉROPORT POUR LA MODÉLISATION DE LA PHRASE 10 ?



...

### RÉPONSE 3-3

La question est moins triviale qu'il n'y paraît au premier abord... Tout dépend en effet de la définition exacte que l'on prête au verbe « desservir » dans notre système ! En effet, si « desservir » consiste simplement à désigner le moyen de transport par les airs le plus proche, toute ville est desservie par un et un seul aéroport.

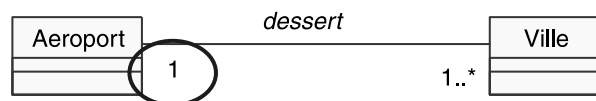


Figure 3-11. Modélisation possible de la phrase 10

Mais si « desservir » vaut par exemple pour tout moyen de transport aérien se trouvant à moins de trente kilomètres, alors une ville peut être desservie par 0 ou plusieurs aéroports.

C'est cette dernière définition que nous retiendrons.

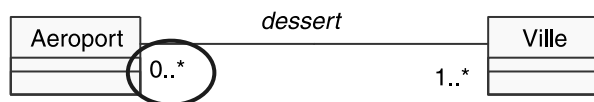


Figure 3-12. Modélisation complétée de la phrase 10

### 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*.

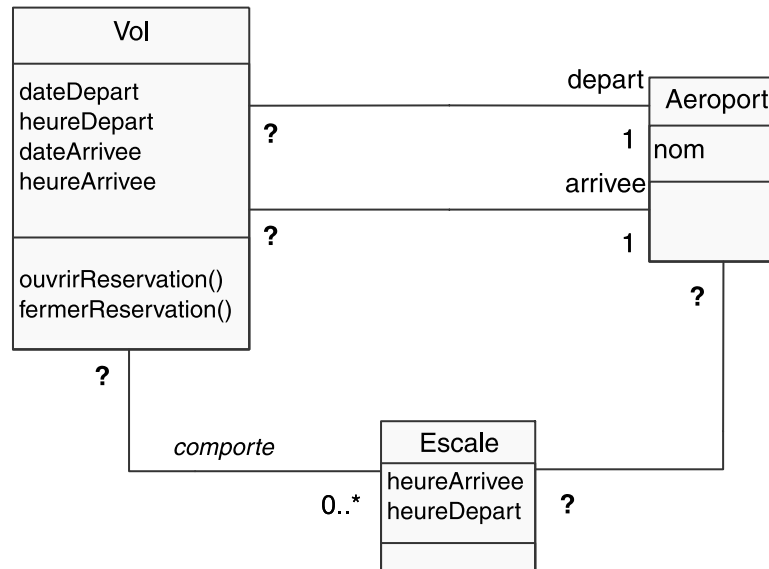


Figure 3-13. Modélisation préliminaire des phrases 8 et 9

**Question \*\***

3-4 COMPLÉTEZ LES MULTIPLICITÉS DES ASSOCIATIONS.



...

**RÉPONSE 3-4**

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.

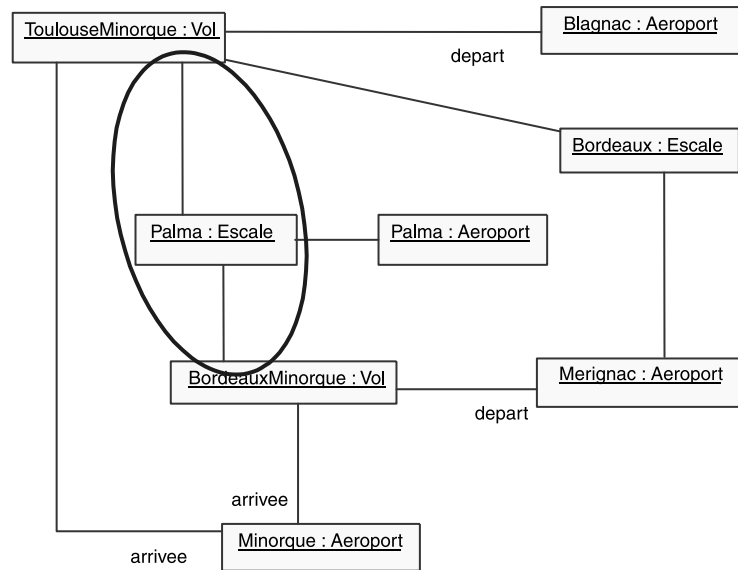


Figure 3-14. Diagramme d'objets illustrant la phrase 8

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.

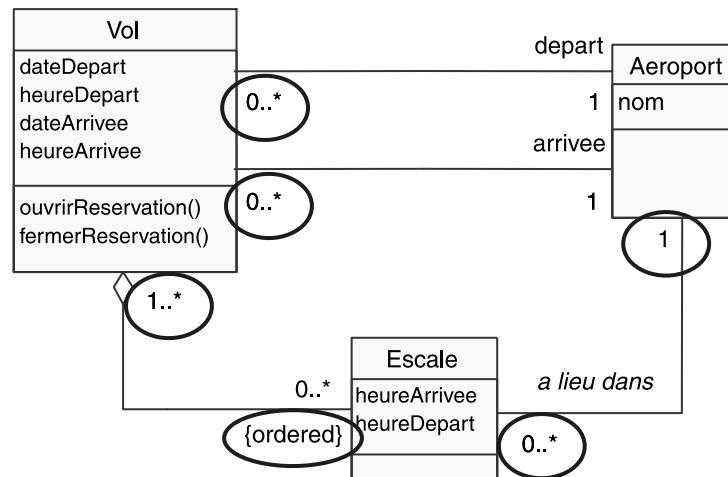


Figure 3-15. Modélisation complète des phrases 8 et 9

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...



3-5 PROPOSEZ UNE SOLUTION PLUS SOPHISTIQUEE POUR LA MODÉLISATION DES ESCALES.

Question\*\*\*



...

### RÉPONSE 3-5

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 *Aéroport*.

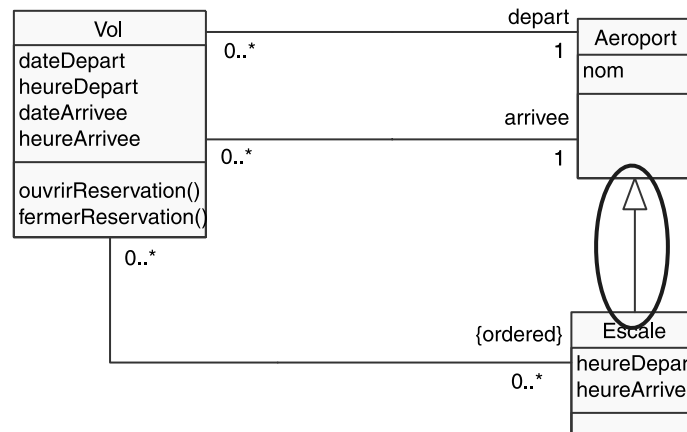


Figure 3-16. Modélisation avec héritage des phrases 8 et 9

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 *Aéroport*, 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.

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*.

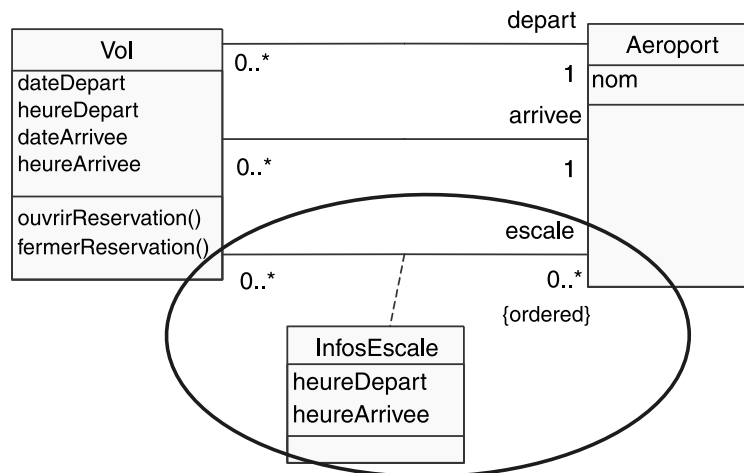


Figure 3-17. Modélisation plus sophistiquée des phrases 8 et 9

## 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.



3-6 FAUT-IL VRAIMENT DISTINGUER LES CONCEPTS DE CLIENT ET DE PASSAGER ?



...

### RÉPONSE 3-6

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.

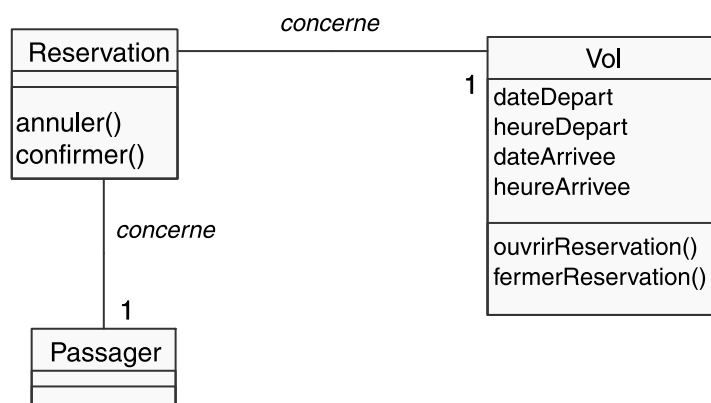


Figure 3-18. Modélisation directe des phrases 4 et 5



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.

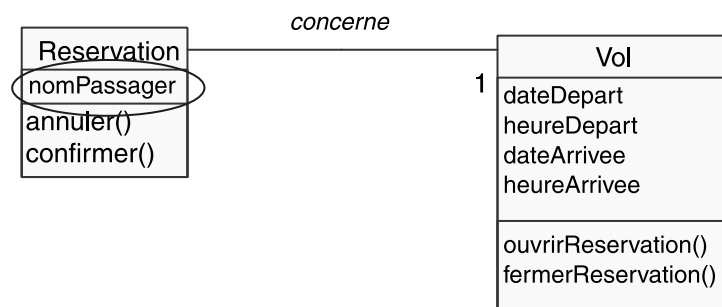



Figure 3-19. Modélisation simpliste des phrases 4 et 5

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

Poursuivons notre traitement du concept de réservation. La phrase 3 est un peu plus délicate, à cause de sa formulation alambiquée.



**Question \*\***

3-7 MODÉLISEZ LA PHRASE 3 ET COMPLÉTEZ LES MULTIPLICITÉS DES ASSOCIATIONS PRÉCÉDENTES.



...

## RÉPONSE 3-7

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.

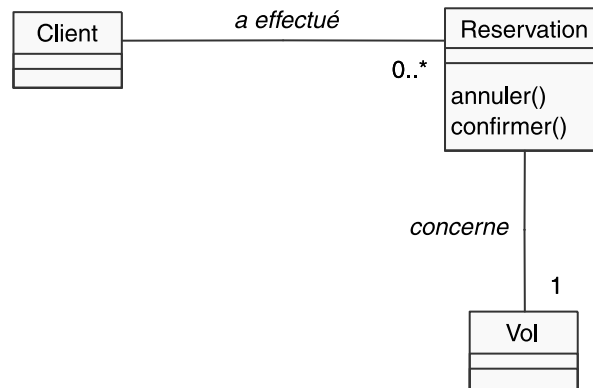


Figure 3-20. Début de modélisation de la phrase 3

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..\* ».

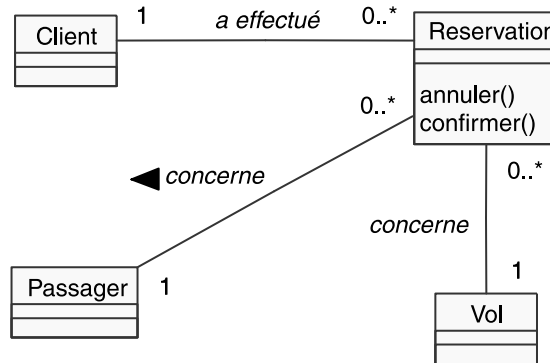


Figure 3-21. Modélisation complète des phrases 4 et 5

## Étape 5 – Ajout d’attributs, de contraintes et de qualificatifs

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.

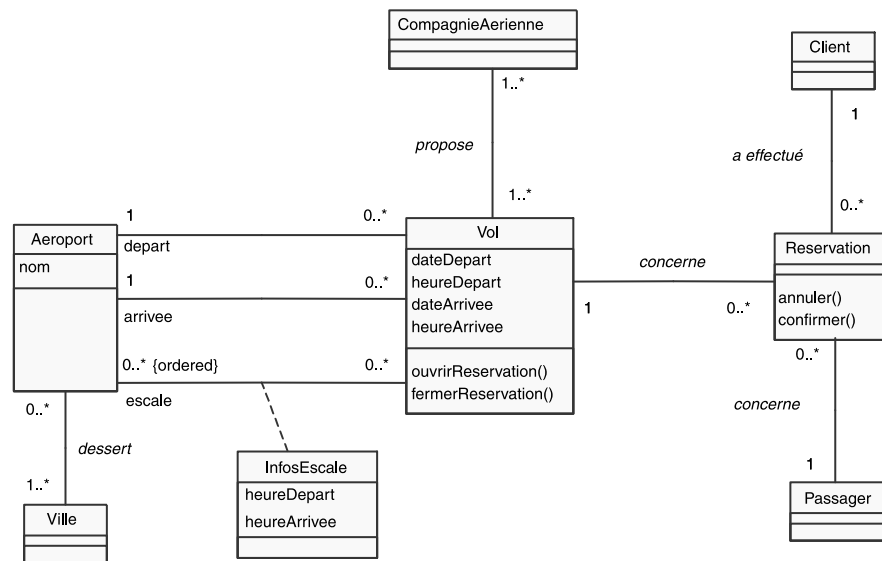


Figure 3-22. Modélisation préliminaire de l’étude de cas 3

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...

**Question \***

3-8 AJOUTEZ LES ATTRIBUTS MÉTIER QUI VOUS SEMBLent  
INDISPENSABLES.



...

---

**RÉPONSE 3-8**

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'identification des associations.

*Aéroport :*

- 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.



#### 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.

---



3-9 COMPLÉTEZ LE MODÈLE AVEC QUELQUES ATTRIBUTS DÉRIVÉS PERTINENTS.



...

### RÉPONSE 3-9

Un attribut dérivé est une propriété évaluée 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.

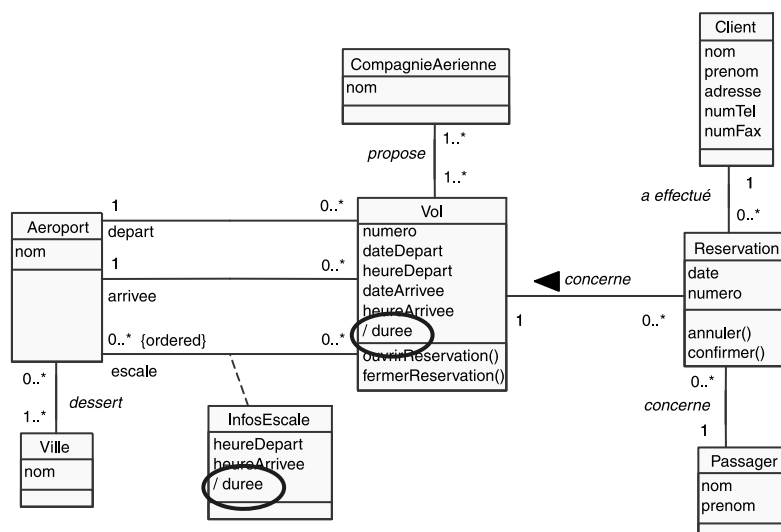


Figure 3-23. Ajout des attributs métier et dérivés



### Nota

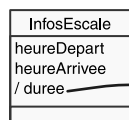
#### ATTRIBUT DÉRIVÉ EN CONCEPTION

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 :

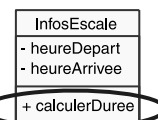
- Garder un attribut « normal » en conception, qui aura ses méthodes d'accès (*get* et *set*) comme les autres attributs : il ne faut pas oublier de déclencher la mise à jour de l'attribut dès qu'une information dont il dépend est modifiée ;
- Ne pas stocker la valeur redondante, mais la calculer à la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



*Exemple de transformation d'un attribut dérivé en méthode de conception*



### Question\*\*\*

3-10 AFFINEZ ENCORE LE DIAGRAMME EN AJOUTANT DES CONTRAINTES ET UNE ASSOCIATION QUALIFIÉE.



...

## RÉPONSE 3-10

Il est possible que l'on trouve un grand nombre de contraintes sur un diagramme de classes. Mieux vaut les répertorier de façon exhaustive dans le texte qui accompagne le modèle, et choisir avec soin les plus importantes, que l'on pourra alors faire figurer sur le diagramme. On encourt sinon le risque de surcharger le schéma et de le rendre illisible.

Sur notre exemple, nous avons choisi de montrer les contraintes les plus fortes entre les attributs. Elles correspondent à des règles de gestion qui devront être implémentées dans le système informatique final.

Nous avons également insisté sur le fait qu'une réservation concerne bien un seul vol et un seul client, et qui plus est de façon irréversible. Pour changer de vol ou de client, il faut annuler la réservation en question, et en créer une nouvelle. Cela se traduit en UML par les contraintes `{frozen}` sur les rôles des associations concernées.

Enfin, nous avons transformé l'attribut *numero* de la classe *Vol* en qualificatif de l'association *propose* entre les classes *CompagnieAerienne* et *Vol*. En effet, chaque vol est identifié d'une façon unique par un numéro propre à la compagnie. Notez comme l'ajout du qualificatif réduit la multiplicité du côté de la classe *Vol*. La figure présentée ci-après montre le diagramme de classes complété.

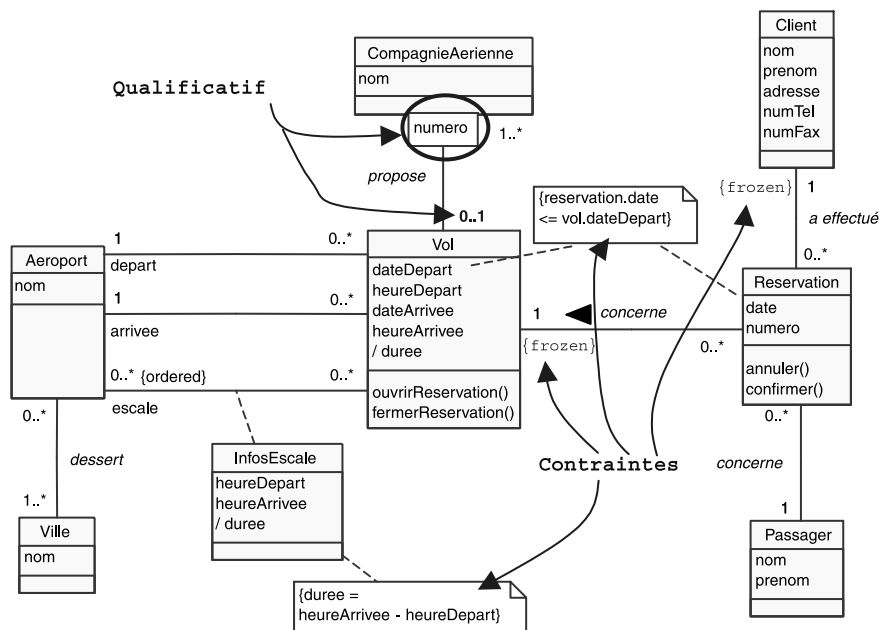


Figure 3-24. Ajout des contraintes et du qualificatif



## Étape 6 – Utilisation de patterns d'analyse

Notre modèle peut encore être notablement amélioré !

Reprenons à cet effet par le détail les éléments qui concernent la classe *Vol*, tels qu'ils sont représentés sur la figure suivante.

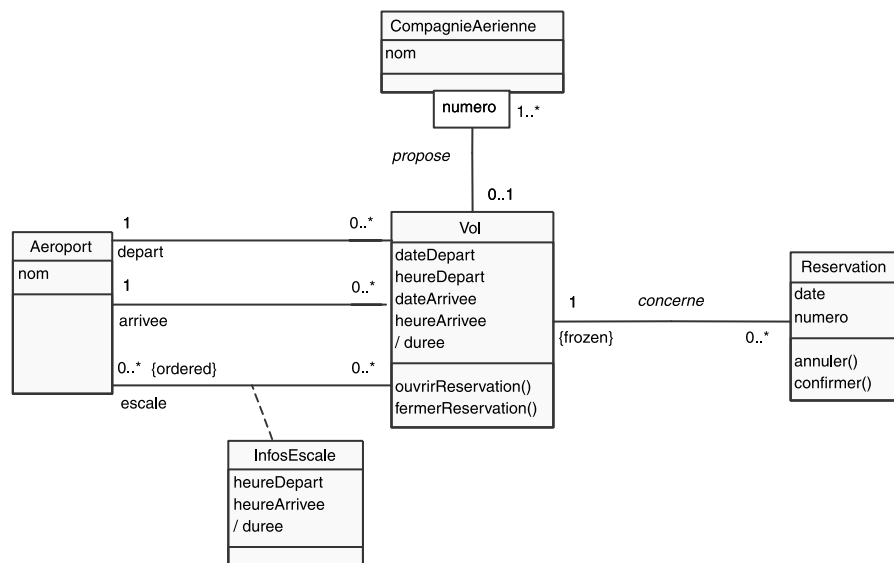


Figure 3-25. Détail du modèle autour de la classe *Vol*

Ne vous semble-t-il pas que la classe *Vol* a beaucoup de responsabilités différentes, avec tous ses attributs et toutes ses associations ? Elle viole un principe fort de la conception orientée objet, appelé par certains auteurs, *forte cohésion* (cf. [Larman 97]).



3-11 PROPOSEZ UNE SOLUTION PLUS SOPHISTIQUÉE POUR LA MODÉLISATION DES VOLS.



...

### RÉPONSE 3-11

La classe *Vol* du diagramme précédent possède deux différents types de responsabilités :

- Le premier concerne tout ce qui se retrouve dans les catalogues des compagnies aériennes : oui, il existe bien un Toulouse-Paris Orly sans escale, tous les lundis matin à 7 h 10, proposé par Air France... Il s'agit là de vols génériques, qui reviennent à l'identique, toutes les semaines, ou presque.
- Le second rassemble ce qui touche aux réservations. Vous ne réservez pas un Toulouse-Paris Orly du lundi matin, mais bien le Toulouse-Paris Orly du 25 juillet 2000 !

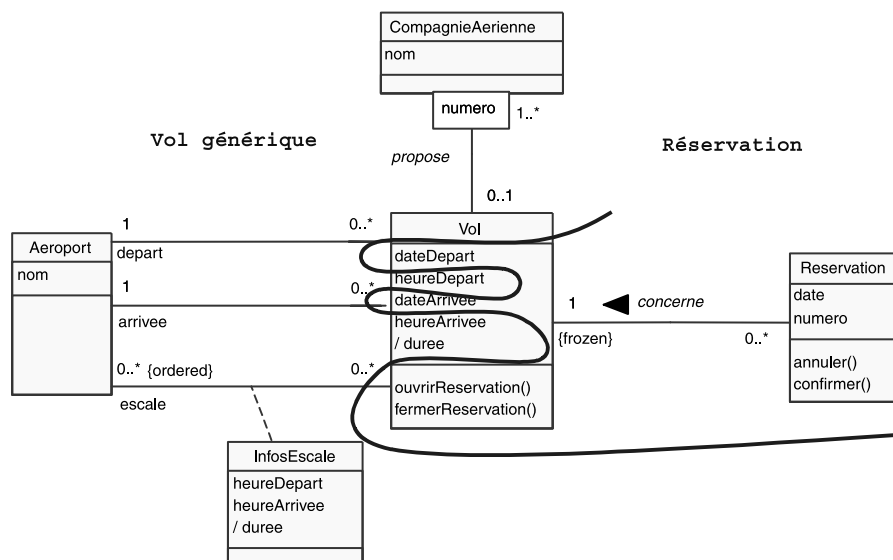


Figure 3-26. Séparation des responsabilités de la classe *Vol*

On peut y voir une sorte de relation d'instanciation, entre une classe *VolGenerique* restreinte au premier type de responsabilités, et une classe *Vol* qui rassemble les responsabilités du second type. En effet, un vol générique décrit une fois pour toutes des propriétés qui seront identiques pour de nombreux vols réels.

De même, supposons qu'une compagnie annule tous ses prochains vols du week-end au départ de l'aéroport X, car celui-ci est indisponible jusqu'à nouvel ordre, en raison d'importants travaux de maintenance effectués le samedi et le dimanche. Cela signifie dans notre première solution que nous allons détruire toutes les instances correspondantes de la classe *Vol*. À la fin de la période de maintenance de l'aéroport X, il nous faudra recréer les instances de *Vol* avec leurs attributs valorisés et leurs liens, à partir de rien. Alors que, si nous comptons une classe *VolGenerique*, les valeurs des attributs et les liens des vols partant de X ne sont pas perdus ; il n'y aura simplement pas d'instance de *Vol* réel correspondante pendant trois mois.

Pour mettre à jour le modèle, il suffit donc de :

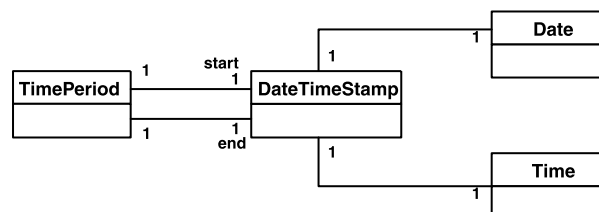
- répartir les attributs, les opérations et les associations de l'ancienne classe *Vol* entre les deux classes *VolGenerique* et *Vol* ;
- ajouter une association « 1-\* » *décrit* entre *VolGenerique* et *Vol*.

Nous avons en outre ajouté deux attributs dans la classe *VolGenerique* pour indiquer le jour de la semaine du vol, et la période de l'année où il est disponible. Une contrainte supplémentaire lie les valeurs des attributs *dateDepart* de la classe *Vol* et de la classe *VolGenerique*.



#### PERIODE VALIDE AVEC UN TYPE NON-PRIMITIF

L'attribut *periodeValidite* n'est pas un attribut simple. En effet, on peut lui demander son début, sa fin, sa durée, etc. Une solution a été proposée par M. Fowler<sup>a</sup> : créer une classe utilisateur *TimePeriod* (comme pour *Date* précédemment), et l'utiliser ensuite pour typer l'attribut.



Modélisation du type « non-primitif » *TimePeriod*

a. *Analysis Patterns: Reusable Object Models*, M. Fowler, 1997, Addison-Wesley.

Enfin, il faut veiller à bien respecter la phrase 2. Un vol est ouvert ou fermé à la réservation sur ordre de la compagnie. C'est bien le vol daté, et pas le vol générique, qui est concerné. Même chose pour une éventuelle annulation... Il nous faut donc ajouter une association directe entre *Vol*

et *CompagnieAerienne*, qui permette l'interaction décrite à la figure 3-5, tout en conservant l'association qualifiée entre *VolGenerique* et *CompagnieAerienne*.

La figure 3-26 est donc transformée comme cela est montré sur le schéma ci-après. Chacune des deux classes – *Vol* et *VolGenerique* – a retrouvé une forte cohésion.

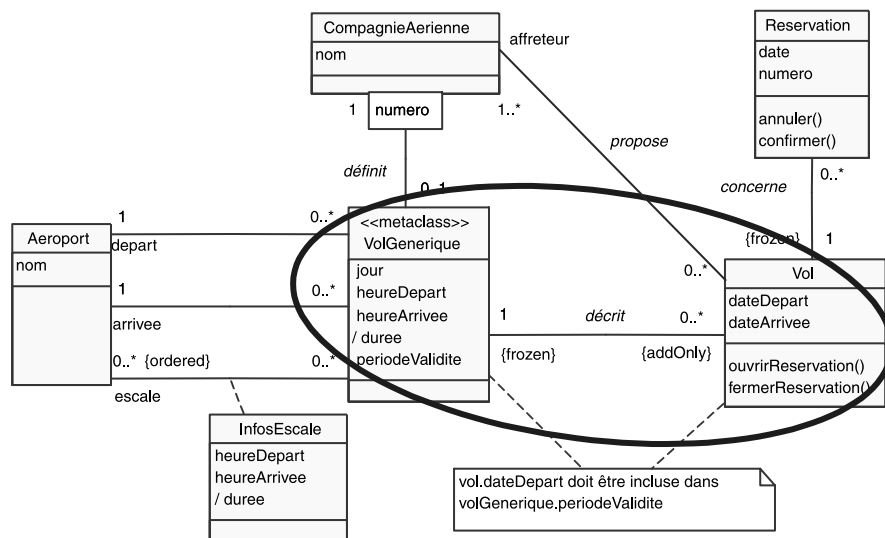


Figure 3-27. Répartition des responsabilités entre *VolGenerique* et *Vol*



#### PATTERN DE LA MÉTA-CLASSE

La séparation des responsabilités opérée précédemment peut se généraliser sous la forme d'un « pattern d'analyse » réutilisable dans d'autres contextes et déjà décrit par de nombreux auteurs.

On identifie une classe *XX* qui possède trop de responsabilités, dont certaines ne sont pas propres à chaque instance. On ajoute une classe *TypeXX*, on répartit les propriétés sur les deux classes et on les relie par une association « \* - 1 ». La classe *TypeXX* est qualifiée de « métaclasse », comme *VolGenerique* sur la figure ci-dessus, car elle contient des informations qui décrivent la classe *XX*.

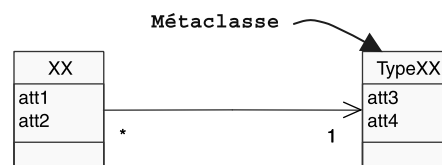


Schéma générique réutilisable

## Étape 7 – Structuration en packages

Notre modèle conceptuel est maintenant presque finalisé. Pour rendre son emploi plus aisé encore et afin de préparer l'activité de conception objet, nous allons le structurer en packages.



### STRUCTURATION EN PACKAGES

La structuration d'un modèle conceptuel est une activité délicate. Elle doit s'appuyer sur deux principes fondamentaux : *cohérence* et *indépendance*.

Le premier principe consiste à regrouper les classes proches d'un point de vue sémantique. Pour cela, il faut que les critères de cohérence suivants soient réunis :

- finalité : les classes doivent rendre des services de même nature aux utilisateurs ;
- évolution : on isole ainsi les classes réellement stables de celles qui vont vraisemblablement évoluer au cours du projet, ou même par la suite. On distingue notamment les classes *métier* des classes *applicatives* ;
- cycle de vie des objets : ce critère permet de distinguer les classes dont les objets ont des durées de vie très différentes.

Le second principe consiste à renforcer ce découpage initial en s'efforçant de minimiser les dépendances entre les packages.

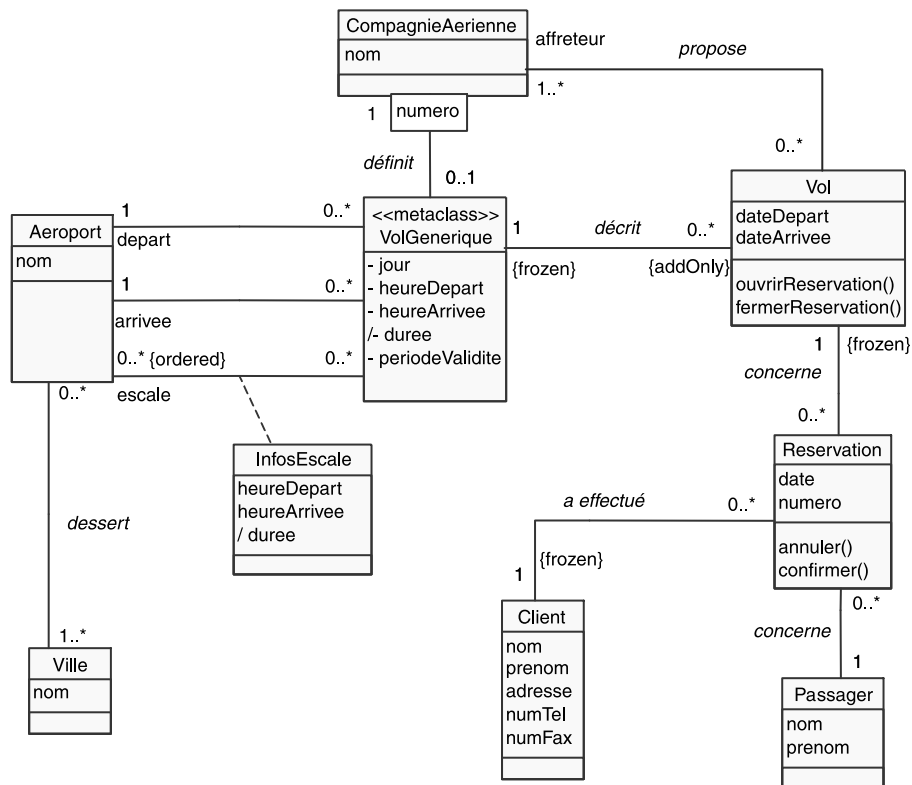


Figure 3-28. Modèle conceptuel avant structuration



## Question \*\*

3-12 PROPOSEZ UN DÉCOUPAGE DU MODÈLE CONCEPTUEL EN DEUX PACKAGES.



...

### RÉPONSE 3-12

D'après les critères précédents, nous pouvons proposer une première découpe en deux packages :

- le premier va concerner la définition des vols, très stable dans le temps, surtout la partie spécifique à *VolGenerique* ;
- le second va traiter des réservations, avec toutes leurs associations.

Chaque package contient bien un ensemble de classes fortement reliées, mais les classes des deux packages sont presque indépendantes. Cette première découpe est matérialisée par la ligne qui partage le schéma présenté ci-après.

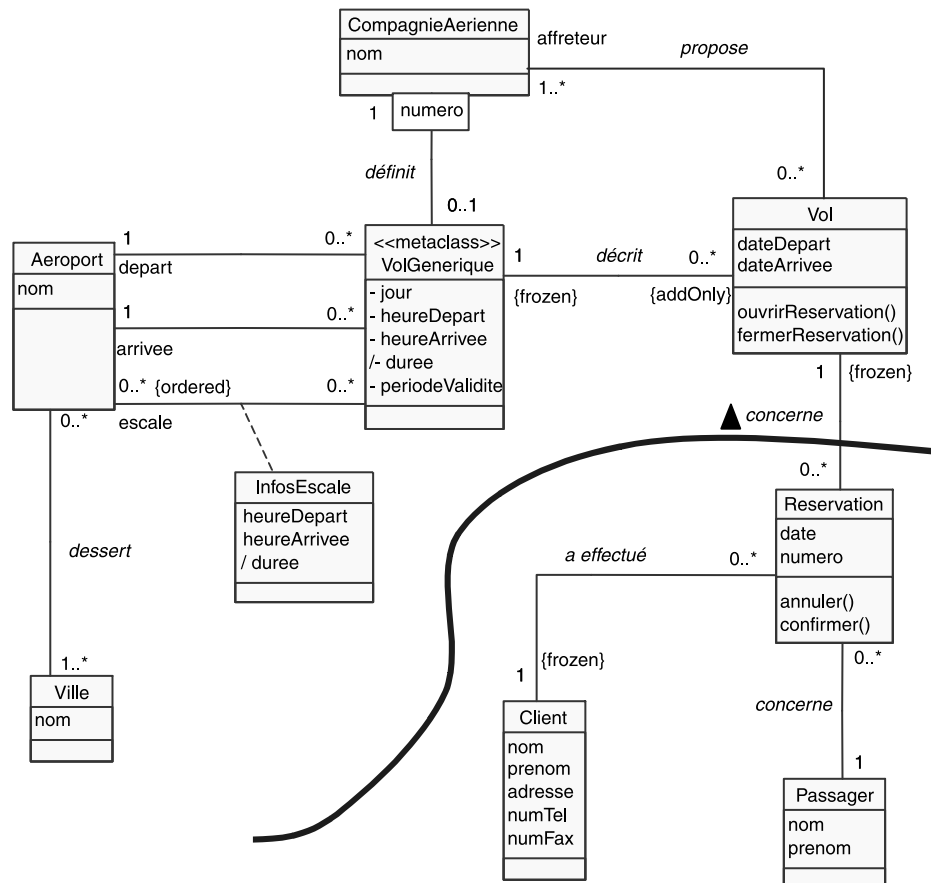


Figure 3-29. Découpe du modèle en deux parties peu dépendantes







Question\*\*\*

3-13 TROUVEZ UNE SOLUTION QUI PERMETTE DE MINIMISER LE COUPLAGE ENTRE LES DEUX PACKAGES.



...

### RÉPONSE 3-13

Dans les deux cas précédents, nous pouvons constater qu'au moins une association traverse la frontière entre les packages. Le problème que posent les associations qui traversent deux packages réside en ceci qu'une seule d'entre elles suffit à induire une dépendance mutuelle, si elle est bidirectionnelle. Or, le concepteur objet doit faire la chasse aux dépendances mutuelles ou cycliques, afin d'augmenter la modularité et l'évolutivité de son application.

Dans la première solution, une seule association est en cause, comme cela est rappelé ci-après. Mais, à elle toute seule, elle provoque une dépendance mutuelle entre les deux packages.

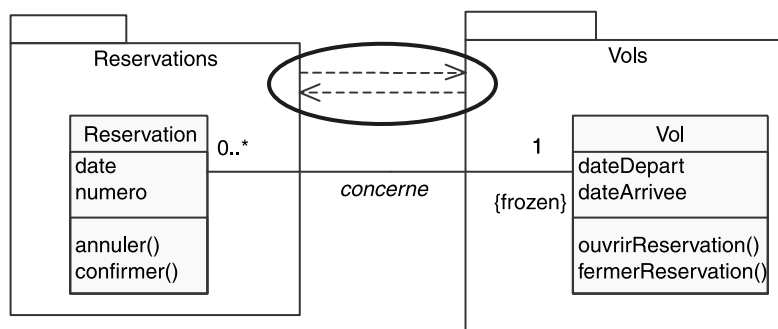


Figure 3-31. Dépendance mutuelle entre les packages



## NAVIGABILITÉ ET DÉPENDANCE

Une association entre deux classes A et B permet par défaut de naviguer dans les deux sens entre des objets de la classe A et des objets de la classe B.

Cependant, il est possible de limiter cette navigation à une seule des deux directions, pour éliminer une des deux dépendances induites par l'association. UML nous permet de représenter explicitement cette navigabilité en ajoutant sur l'association une flèche qui indique le seul sens possible.

Dans notre exemple, nous allons faire un choix et privilégier un sens de navigation afin d'éliminer une des deux dépendances. Il est certain qu'une réservation ne va pas sans connaissance du vol qui est concerné, alors qu'un vol existe par lui-même, indépendamment de toute réservation.

Le diagramme précédent peut donc être modifié de façon qu'il ne fasse apparaître que la dépendance du package *Reservations* vers le package *Vols*.

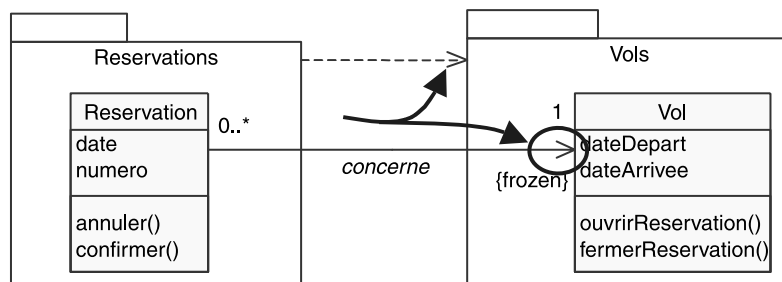


Figure 3-32. Couplage minimisé entre les packages

Examinons maintenant la seconde solution. Cette fois-ci, deux associations traversent les packages. Que pouvons-nous faire pour réduire les navigabilités de ces associations ?

Il est logique d'imposer un sens de navigabilité unique de *Vol* vers *VolGenerique* : un vol réel est décrit par un et un seul vol générique auquel il doit avoir accès, alors qu'un vol générique peut exister par lui-même.

Hélas, pour la seconde association, nous savons déjà que la navigabilité est obligatoire de *CompagnieAerienne* vers *Vol*, à cause de la phrase 2, illustrée par le diagramme de collaboration de la figure 3-4.

Même si nous enlevons la navigabilité de *Vol* vers *CompagnieAerienne*, nous nous retrouvons donc avec deux associations navigables dans des sens différents. Cela suffit à imposer une dépendance mutuelle entre les packages, comme cela est représenté ci-après.

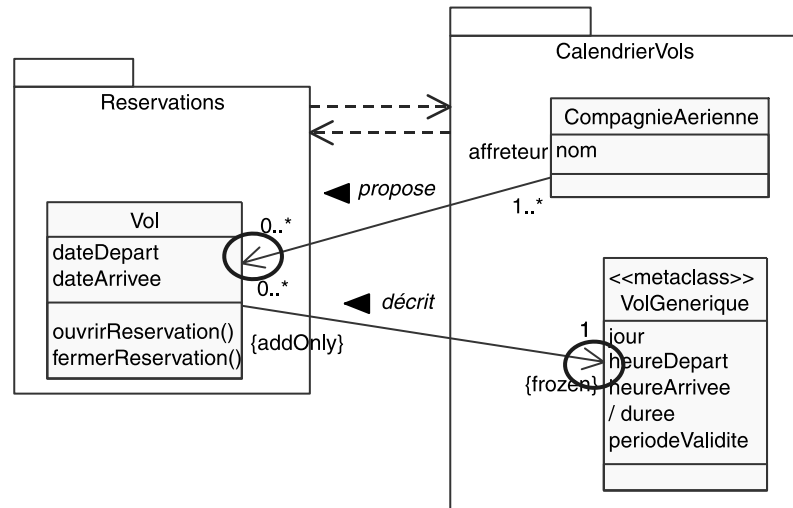


Figure 3-33. Dépendance mutuelle obligatoire pour la seconde solution

Cette étude sur le couplage des packages pour les deux solutions proposées fait donc pencher la balance vers la première solution, ce qui n'était pas du tout évident au départ.

Le package *Vols* peut maintenant se prêter à une réutilisation, contrairement au package *Reservations*.

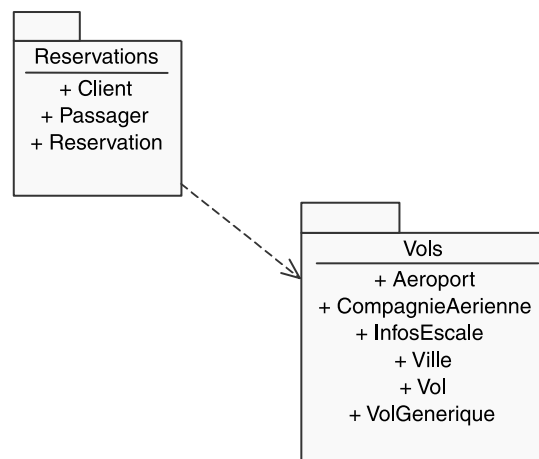


Figure 3-34. Diagramme structurel de packages de la solution retenue

L'état complet de notre modèle peut maintenant être synthétisé par le schéma suivant.

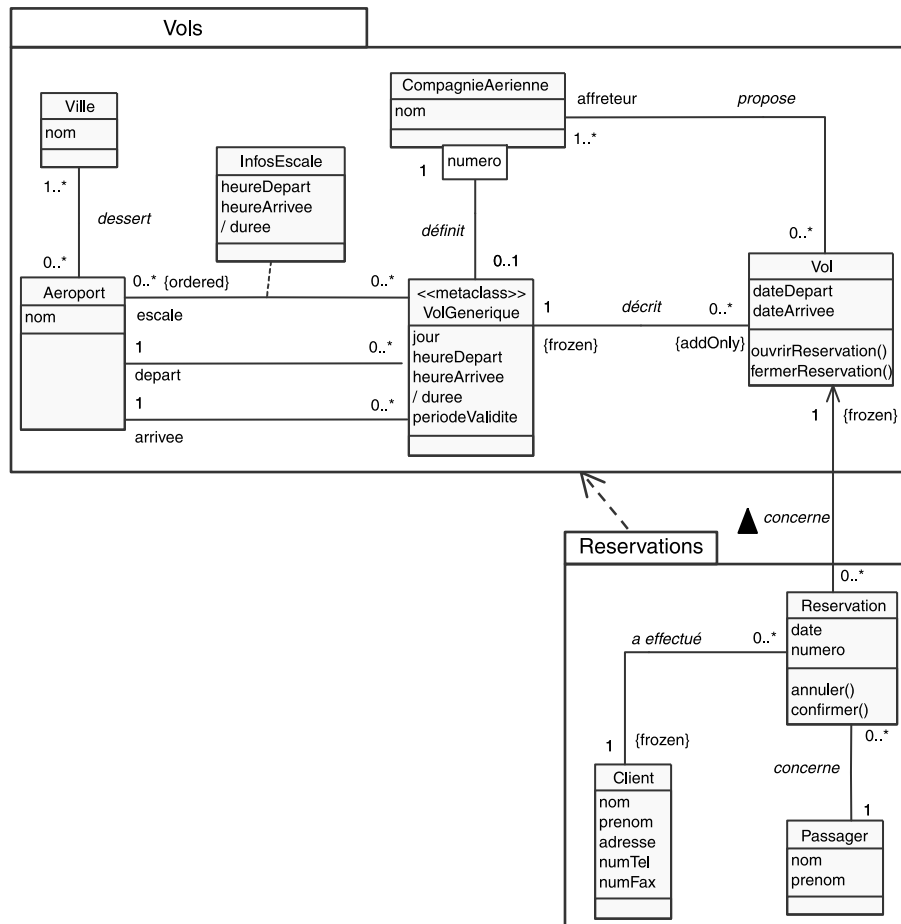


Figure 3-35. Modèle complet de la réservation de vols

## Étape 8 – Généralisation et réutilisation

Après tout ce travail sur les réservations de vols, nous souhaitons élargir le champ du modèle en proposant également des voyages en bus, que des transporteurs assurent.

Un voyage en bus a une ville de départ et une ville d'arrivée, avec des dates et heures associées. Le trajet peut comporter des arrêts dans des villes intermédiaires.

Un client peut réserver un ou plusieurs voyages, pour un ou plusieurs passagers.



Question\*\*\*

3-14 PAR ANALOGIE AVEC LA FIGURE PRÉCÉDENTE, PROPOSEZ UN MODÈLE DE LA RÉSERVATION DE VOYAGES EN BUS.



...

---

### RÉPONSE 3-14

Le modèle est quasiment la réplique du précédent, y compris au niveau de la découpe en packages.

Il est un peu plus simple pour deux raisons :

- la notion d'aéroport n'a pas d'équivalent, et la classe *Ville* est directement associée à la classe *VoyageEnBus* ;
- la distinction entre *Vol* et *VolGenerique* ne semble pas transposable, car les voyages en bus ne sont pas aussi réguliers et ne sont pas définis à l'avance.

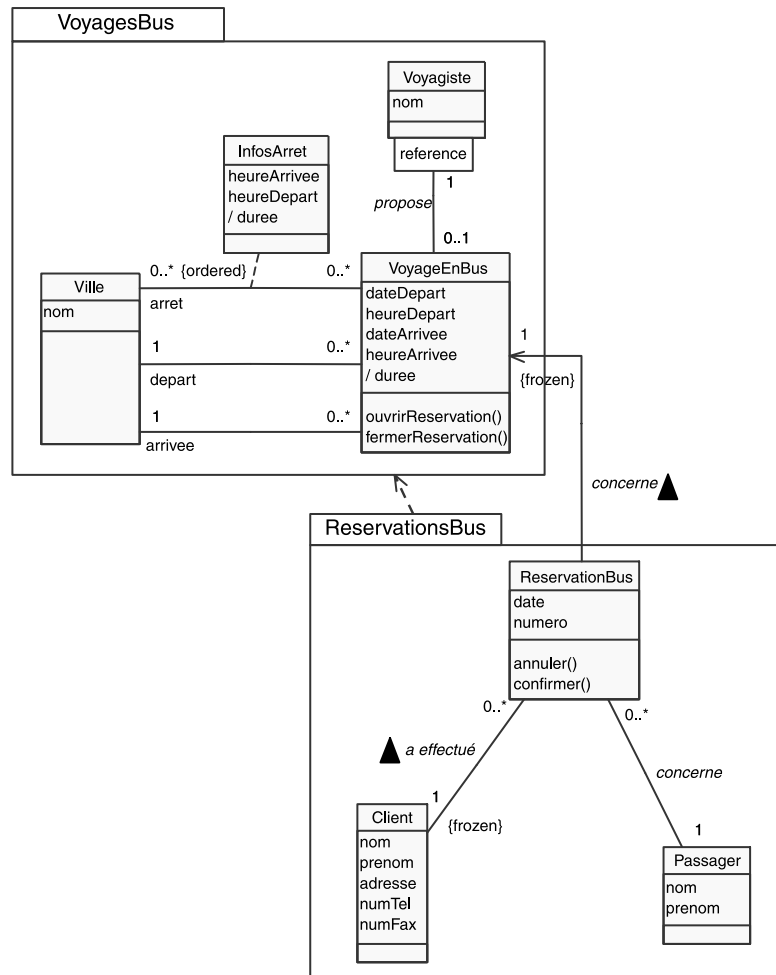


Figure 3-36. Modèle de la réservation de voyages en bus

Il est clair que les diagrammes des figures 3-35 et 3-36 présentent de nombreuses similitudes :

- certaines classes sont communes aux deux modèles : *Ville*, *Client*, *Passager* ;
- certaines classes ont des points communs : *ReservationBus* et *Reservation*, *InfosEscale* et *InfosArret*, etc.

Nous allons donc essayer de faire fusionner ces deux modèles en factorisant autant que faire se peut les concepts, afin de pouvoir encore élargir sa portée si nécessaire (réservation de croisières, etc.).

**Question\*\*\***

3-15 PROPOSEZ UNE ARCHITECTURE LOGIQUE FUSIONNÉE QUI SOIT LA PLUS ÉVOLUTIVE POSSIBLE .



...

---

**RÉPONSE 3-15**

Deux tâches principales doivent être réalisées :

- Isoler les classes communes dans de nouveaux packages, afin de pouvoir les réutiliser
- Factoriser les propriétés communes dans des classes abstraites.

En premier lieu, commençons par l'identification et le regroupement des classes communes.

La classe *Ville* est très importante pour la description des vols et des voyages en bus. Dans le modèle de la figure 3-39, nous avons en fait réutilisé la classe *Ville* existante, ce qui a immédiatement créé une dépendance non justifiée entre les packages *VoyagesBus* et *Vols*. Au lieu de procéder ainsi, il serait plus pertinent de l'isoler dans un package à part qui pourra être réutilisé à volonté, voire même qui pourra être acheté dans le commerce, avec son instanciation par pays...

Afin que ce nouveau package soit vraiment un composant réutilisable, il ne faut pas qu'il dépende des packages applicatifs qui contiennent les classes *Aéroport* et *VoyagesBus*. Pour ce faire, nous avons déjà vu qu'il suffisait d'agir sur la navigabilité des associations concernées, comme cela est indiqué sur le schéma ci-après.

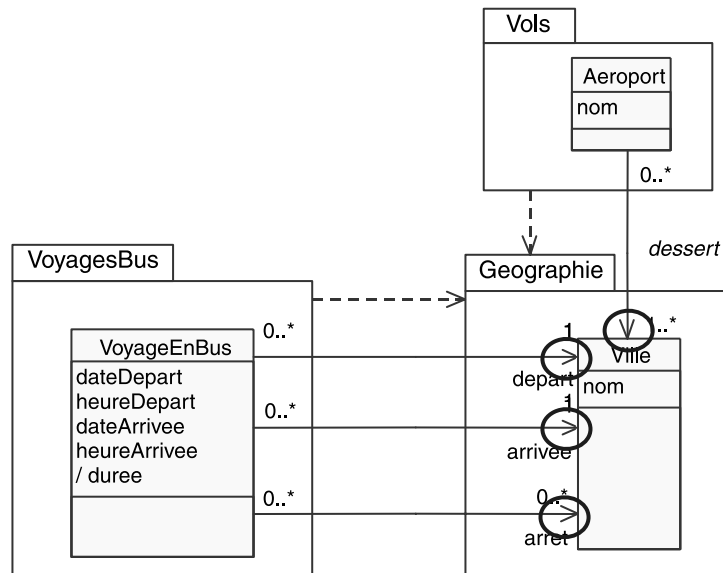


Figure 3-37. Isolation de la classe `Ville` dans un nouveau package réutilisable

Les classes `Client` et `Passager` sont également communes aux deux types de réservations. Nous aurons donc intérêt à les isoler dans un nouveau package, comme cela a été fait pour la classe `Ville`. Mais il ne serait pas judicieux de regrouper ces trois classes communes dans le même package, simplement parce qu'elles sont communes. En effet, les concepts qu'elles représentent n'ont pas de rapport...

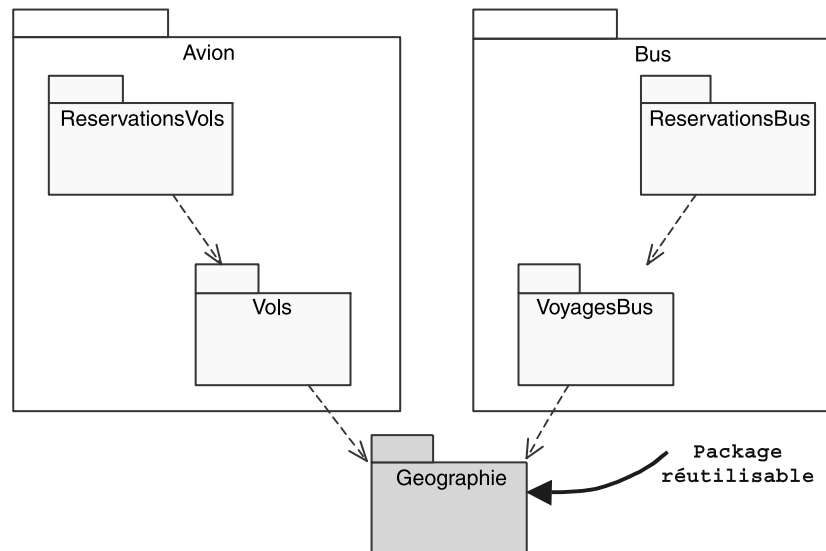


Figure 3-38. Identification du package réutilisable `Géographie`



Après cette première tâche d'isolation des classes communes réutilisables, l'architecture logique se présente sous la forme du diagramme structurel qui précède.

Il nous faut maintenant factoriser les parties communes.

Commençons par ce qui est le plus visible : la similitude entre les packages *ReservationsVols* et *ReservationsBus* saute aux yeux. La seule différence concerne les classes *ReservationVol* (appelée précédemment *Reservation* et renommée pour plus de clarté) et *ReservationBus* : elles ont les mêmes attributs et opérations, et presque les mêmes associations. Une super-classe abstraite *Reservation* s'impose donc en toute logique, comme cela est illustré sur le schéma suivant.

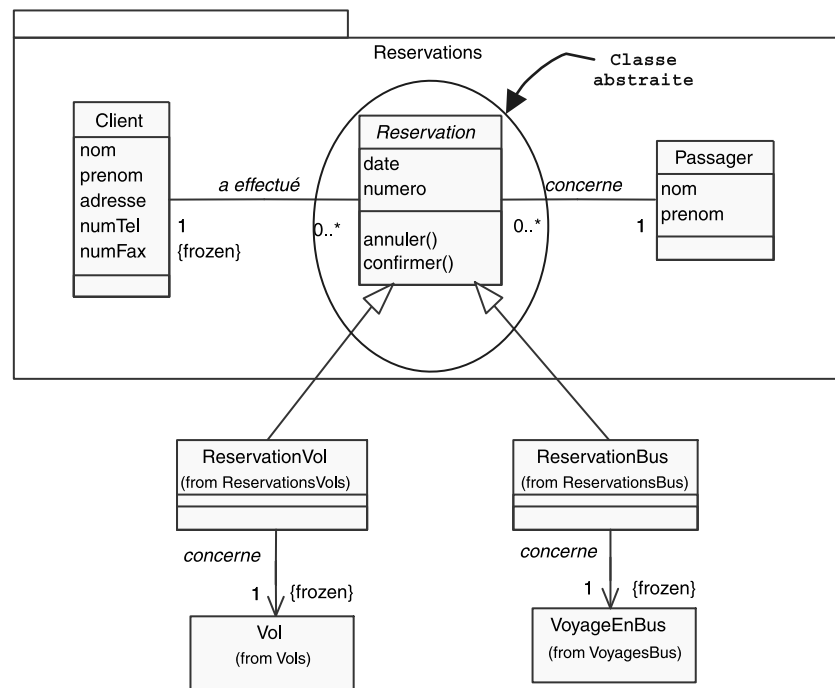


Figure 3-39. Introduction de la classe abstraite *Reservation* par généralisation

La classe abstraite *Reservation* ainsi que les deux classes *Client* et *Passager*, qui sont communes aux deux types de moyen de transport, sont isolées dans un nouveau package appelé *Reservations*. Ce package est appelé package généralisé en regard des deux packages *ReservationsVols* et *ReservationsBus*. En effet, les deux packages spécialisés héritent les classes de *Reservations*, et ont le droit d'en redéfinir certaines.

Le schéma global des packages qui sont ainsi obtenus est représenté sur la figure suivante.

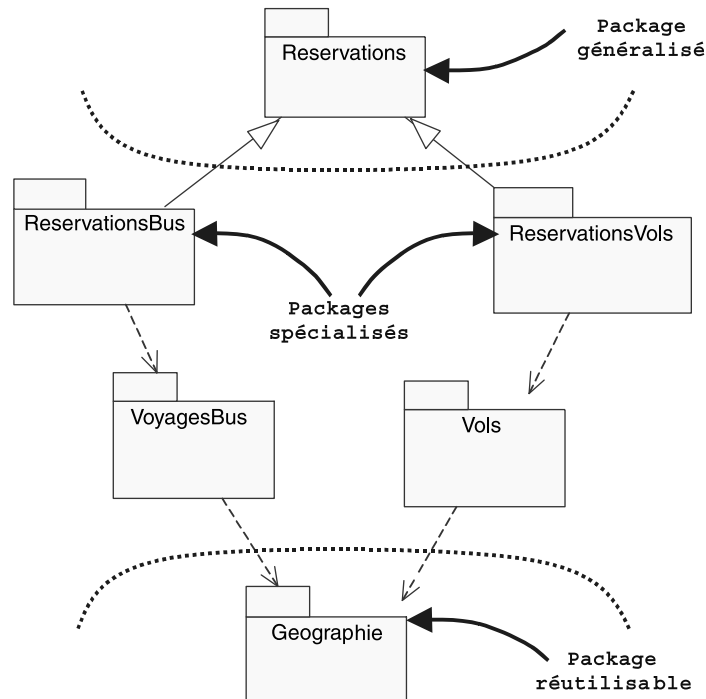


Figure 3-40. Introduction du package généralisé Reservations

## Bibliographie

- [Coad 97] *Object Models: Strategies, Patterns and Applications*, 2<sup>nd</sup> ed., P. Coad, D. North, M. Mayfield, 1997, Prentice Hall.
- [Fowler 97] *Analysis Patterns: Reusable Object Models*, M. Fowler, 1997, Addison-Wesley.
- [Gamma 95] *Design Patterns: Elements of Reusable Object-Oriented Software*, E. Gamma et al., 1995, Addison-Wesley.
- [Hay 96] *Data Model Patterns: Conventions of Thought*, D. Hay, 1996, Dorset House Publishing.
- [Larman 97] *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design*, C. Larman, 1997, Prentice Hall.
- [Roques 00] *UML en action: de l'analyse des besoins à la conception Java*, P. Roques, F. Vallée, 2000, Eyrolles.
- [Rumbaugh 91] *Object-Oriented Modeling and Design*, J. Rumbaugh et al., 1991, Prentice Hall.