Modélisation statique exercices corrigés et conseils méthodologiques

Mots-clés

■ Agrégation – composition ■ Généralisation – spécialisation ■ Classe abstraite – classe concrète ■ Contrainte ■ Qualificatif ■ Pattern ■ Classe structurée – participant – rôle – connecteur ■ Diagramme de structure composite

Ce chapitre va nous permettre de compléter au moyen de plusieurs petits exercices notre passage en revue des principales difficultés que pose la construction des diagrammes de classes UML.

Nous aborderons en particulier des sujets avancés tels que :

- la distinction entre agrégation et composition ;
- les limitations de la relation de composition ;
- les classes structurées UML 2 et le diagramme de structure composite ;
- la bonne utilisation de la généralisation et des classes abstraites ;
- la bonne utilisation des classes d'association ;
- les contraintes entre associations (*xor*, *subset*, etc.) ;
- d'autres *patterns* d'analyse comme « Party » ou « Composite ».

COMPLÉMENTS SUR LES RELATIONS ENTRE CLASSES

EXERCICE 4-1.
Relations structurelles entre classes

Considérons les phrases suivantes :

1. Un répertoire contient des fichiers.
2. Une pièce contient des murs.
3. Les modems et les claviers sont des périphériques d'entrée/sortie.
4. Une transaction boursière est un achat ou une vente.
5. Un compte bancaire peut appartenir à une personne physique ou morale.
6. Deux personnes peuvent être mariées.

Déterminez la relation statique appropriée (généralisation, composition, agrégation ou association) dans chaque phrase de l'énoncé précédent. Dessinez le diagramme de classes correspondant.

N'hésitez pas à proposer *différentes* solutions pour chaque phrase.

solution

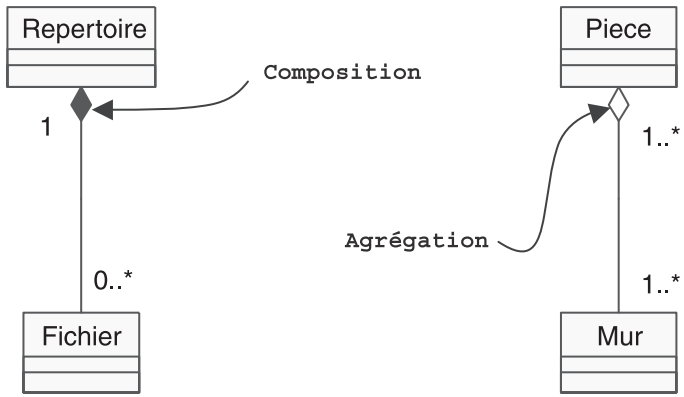
Les phrases 1 et 2 illustrent ce qui différencie l'agrégation de la composition :

1. *Un répertoire contient des fichiers.*
2. *Une pièce contient des murs.*

« Un répertoire contient des fichiers » : il s'agit au moins d'une agrégation. Voyons si nous pouvons aller plus loin et en faire une composition. Premier critère à vérifier : la multiplicité ne doit pas être supérieure à un du côté du composite. C'est bien le cas dans la première phrase, puisqu'un fichier appartient à un et un seul répertoire. Second critère : le cycle de vie des parties doit dépendre de celui du composite. Là encore, c'est le cas, puisque la destruction d'un répertoire entraîne la destruction de tous les fichiers qu'il contient. Nous pouvons donc parler de composition pour la première phrase.

Procédons à la même analyse pour la seconde phrase, « Une pièce contient des murs ». Cette fois-ci, après vérification du premier critère, nous devons abandonner la composition. En effet, un mur peut appartenir à deux pièces contiguës (voire plus). La relation n'est donc qu'une agrégation. Afin de compléter les multiplicités, nous considérons qu'une pièce contient au moins un mur (circulaire !).

Figure 4-1.
Diagramme de classes des phrases 1 et 2



Les phrases 3 et 4 se modélisent en UML par des relations de généralisation :

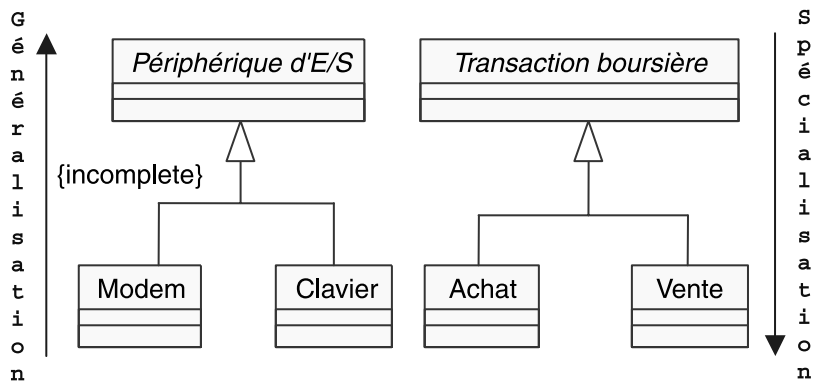
3. Les modems et les claviers sont des périphériques d'entrée/sortie ;

4. Une transaction boursière est un achat ou une vente.

La seule différence réside dans la formulation de la phrase 4 qui correspond à une spécialisation, alors que celle de la phrase 3 est une généralisation. Cependant, nous pouvons apporter quelques précisions aux deux modèles :

- Les super-classes sont abstraites : elles ne s'instancient pas directement, mais toujours par l'intermédiaire d'une de leurs sous-classes ;
- L'arbre de généralisation de la phrase 3 est incomplet : il existe de nombreux autres périphériques d'entrée/sortie, comme les écrans, les souris, etc.

Figure 4-2.
Diagrammes de classes des phrases 3 et 4

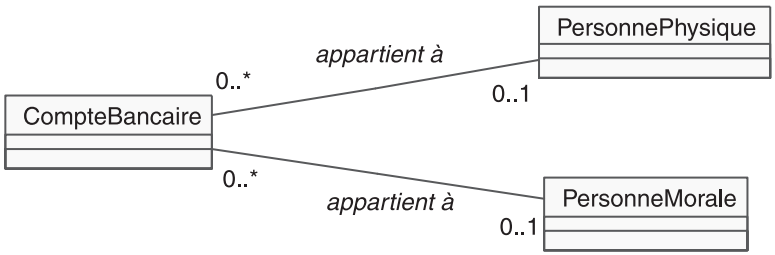


La phrase 5 n'est pas une simple généralisation :

5. Un compte bancaire peut appartenir à une personne physique ou morale.

En effet, le groupe verbal employé n'est pas « est un » ou « est une sorte de », mais « appartient à ». Il s'agit donc d'une simple association. Une première approche simpliste consiste à décrire deux associations optionnelles, comme cela est illustré par la figure suivante.

Figure 4-3.
Diagramme de classes de la phrase 5 – Solution incorrecte

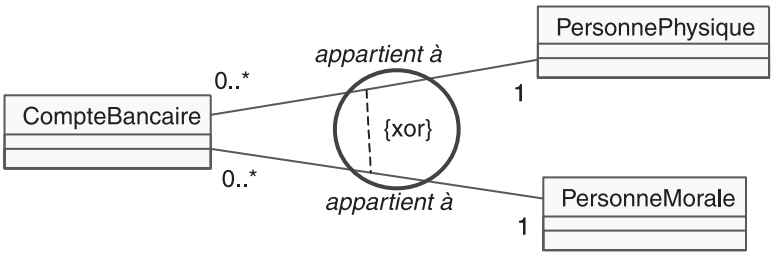


Cette solution ne rend cependant pas compte du « ou exclusif » de la fin de la phrase. En effet, le diagramme précédent peut s'instancier aussi bien avec un objet *CompteBancaire*, lié à la fois à une *PersonnePhysique* et une *PersonneMorale*, qu'avec un *CompteBancaire* lié à aucun objet. Ce n'est pas ce que nous voulons : un objet *CompteBancaire* doit être lié soit à une *PersonnePhysique*, soit à une *PersonneMorale*, pas aux deux à la fois, mais exactement à une des deux, à l'exclusion de l'autre.

En fait, deux solutions correctes mais très différentes sont possibles qui consistent à :

- Introduire explicitement la contrainte prédéfinie {xor} entre les deux associations qui portent une multiplicité strictement égale à 1 ;

Figure 4-4.
Diagramme de classes de la phrase 5 – Première solution



- Introduire une classe abstraite *Personne*, la spécialisation jouant implicitement le rôle du « ou exclusif ».

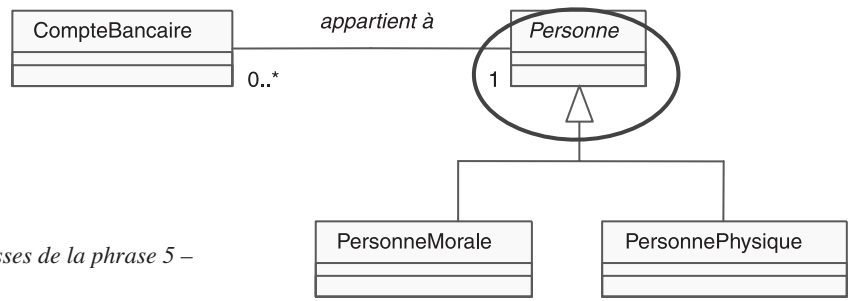


Figure 4-5.
Diagramme de classes de la phrase 5 – Seconde solution

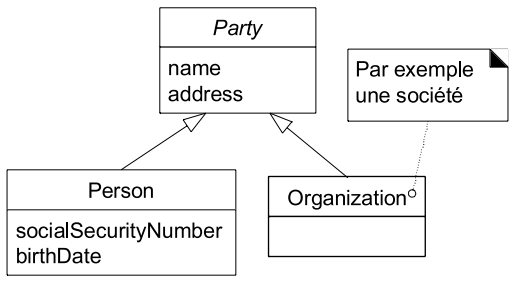
Les deux solutions sont également valables en UML et correctes. Voilà encore un bon exemple de ce que la modélisation n'est pas une science exacte, avec une solution unique pour un problème donné. Le modélisateur a donc le choix entre ces deux schémas.

Un argument important en faveur de la seconde solution est que l'on peut factoriser des attributs (*nom*, *adresse*, etc.) et des opérations (*déménager*, etc.) dans la classe abstraite.

À retenir

LE PATTERN « PARTY »

Cette façon de modéliser des entités qui ont un nom et une adresse uniques (comme les personnes physiques ou morales) par une classe abstraite et deux sous-classes spécialisées a été proposée par D. Hay[a].



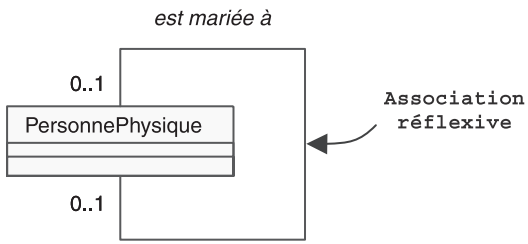
a. *Data Model Patterns : Conventions of Thought*, D. Hay, 1996, Dorset House Publishing.

La phrase 6 présente la particularité de définir une relation entre objets de la même classe.

6. *Deux personnes peuvent être mariées.*

Cela se traduit tout simplement par une association entre cette classe donnée et elle-même. Les multiplicités du schéma suivant sont déduites du droit français actuel : une personne n'est pas tenue d'être mariée, mais ne peut pas être mariée avec plusieurs personnes à la fois !

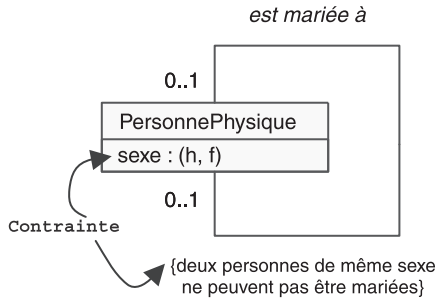
Figure 4-6.
Diagramme de classes de la phrase 6



Si l'on veut ajouter la contrainte que le mariage ne peut unir que des personnes de sexe opposé[1], il y a là encore deux solutions :

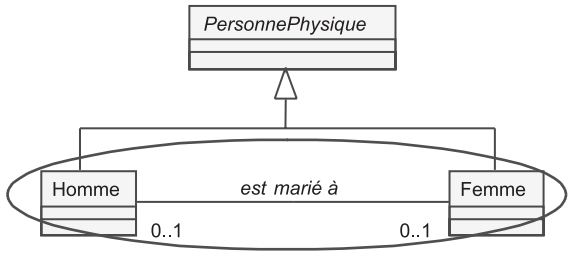
- Introduire explicitement un attribut énuméré *sexe : (h, f)* et une contrainte sur l'association ;

Figure 4-7.
Diagramme de classes complété de la phrase 6



- Introduire deux sous-classes *Homme* et *Femme* comme sur la figure suivante.

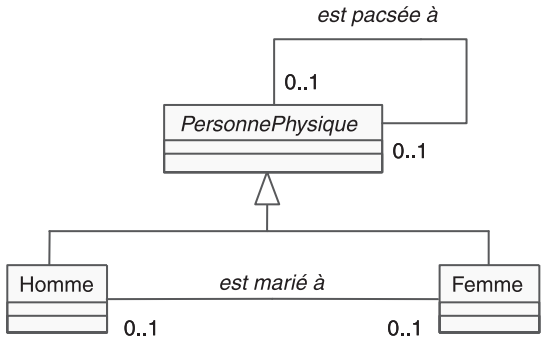
Figure 4-8.
Version du diagramme de classes de la phrase 6 avec spécialisation



1. Bien que cette contrainte soit de plus en plus abandonnée dans les pays européens de nos jours...

Pour compléter le modèle, prenons en compte la nouvelle possibilité offerte par le PACS. Cela amène de nouveau une association réflexive, cette fois-ci non contrainte…

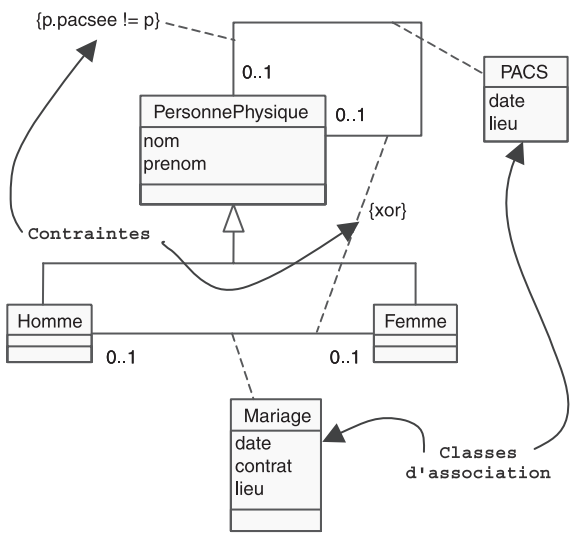
Figure 4-9.
Ajout du dispositif prévu par le Pacs



On notera toutefois qu'il faut en toute rigueur ajouter une contrainte qui interdise d'être à la fois marié avec quelqu'un et « pacsé » avec quelqu'un d'autre, ou « pacsé » avec soi-même…

En outre, l'ajout de la date du mariage, du type de contrat, etc., illustre l'utilisation de la classe d'association. Comme UML interdit à une classe d'association d'avoir à la fois un nom sur l'association et un autre dans la classe, le schéma devient alors :

Figure 4-10.
Version complétée du diagramme de classes de la phrase 6 avec classes d'association



EXERCICE 4-2.
Diagramme de classes : d'un modèle simple, mais peu évolutif, à un modèle souple, mais complexe…

Proposez plusieurs solutions pour modéliser la phrase suivante : « un pays a une capitale. »

Dessinez les diagrammes de classes correspondants et indiquez les avantages et inconvénients des différentes solutions.

solution

Une phrase aussi simple que « un pays a une capitale » va nous permettre d'illustrer le caractère hautement subjectif de l'activité de modélisation, et le choix souvent difficile qu'il faut opérer entre simplicité et évolutivité.

En effet, nous allons proposer pas moins de quatre solutions différentes à cette question, de la plus simple à la plus sophistiquée…

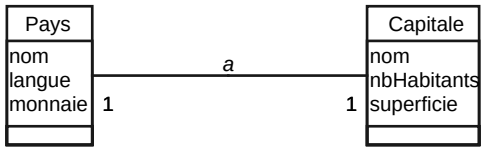
Première solution, la plus compacte possible : une classe Pays avec un simple attribut capitale. C'est suffisant, si nous voulons seulement récupérer le nom de la capitale de chaque pays, et par exemple produire un petit tableau sur deux colonnes, avec les pays classés par ordre alphabétique…

Figure 4-11.
Solution compacte

Difficile de faire plus simple ! Nous pourrons par la suite compléter facilement le modèle en ajoutant quelques attributs à la classe Pays : *nom*, *langue*, *monnaie*, etc.

En revanche, comment faire si nous voulons ajouter des propriétés au concept de capitale : nombre d'habitants, superficie, etc. ? La solution précédente trouve là sa limite, et nous sommes alors obligés de promouvoir capitale au rang de classe. Nous retrouvons ici l'illustration de la différence entre classe et attribut, déjà discutée lors de la question 3-2 du chapitre précédent.

Figure 4-12.
Solution « naturelle »



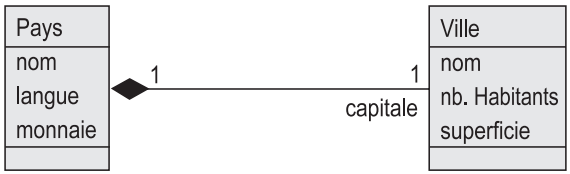
Pour poursuivre cette solution dite « naturelle », on peut se demander à juste titre si l'association n'est pas une agrégation, voire une composition. En effet, un pays contient sa capitale et l'agrégation rappelle la contenance au sens cartographique. Maintenant, peut-on aller plus loin et parler de composition ?

Une capitale appartient bien à un pays et un seul, vérifiant en cela le premier critère de la composition. Cependant, que se passe-t-il en cas de destruction d'un pays ? Si la capitale est également détruite, il s'agit bien d'une composition ; dans le cas contraire, ce n'est qu'une agrégation. Nous touchons là une difficulté venant de ce que la question ne mentionne aucun contexte qui nous permettrait de savoir comment ces concepts de pays et de capitale vont être utilisés. « Détruire » un pays peut aussi bien signifier le conquérir lors d'une guerre que l'enlever de la base de données de notre application informatique… Dans le second cas, la composition ne se discute pas, alors que dans le premier, c'est nettement moins clair. Il faut un peu de réflexion pour comprendre que, là aussi, la capitale disparaît en tant que concept administratif, même si elle n'est pas détruite physiquement. Le schéma se précise donc comme il est indiqué sur la figure suivante.

Figure 4-13.
Solution « naturelle » affinée

En fait, nous sentons bien au fil des phrases qu'un concept plus général que capitale nous fait défaut : la notion de ville. Supposons qu'un pays soit annexé : sa capitale n'existe plus en tant qu'entité administrative, mais la ville elle-même ne sera pas forcément détruite ! Donc, si nous souhaitons définir un modèle plus général, il est intéressant de modéliser le fait qu'une capitale est une ville qui joue un rôle particulier au sein d'un pays. Une première solution incomplète est donnée ci-après.

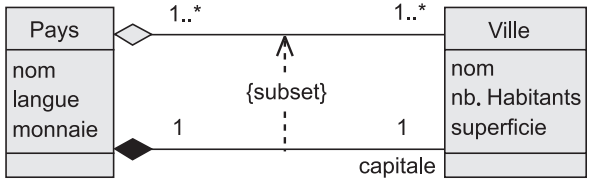
Figure 4-14.
Introduction du concept de ville



Il serait dommage de ne pas profiter de l'introduction du concept plus général de ville pour exprimer le fait qu'un pays contient des villes, dont une seule joue le rôle de capitale. Nous allons donc ajouter une agrégation multiple

entre *Pays* et *Ville*, et une contrainte pour exprimer le fait que la capitale d'un pays est forcément une de ses villes. Le modèle devient maintenant nettement plus sophistiqué…

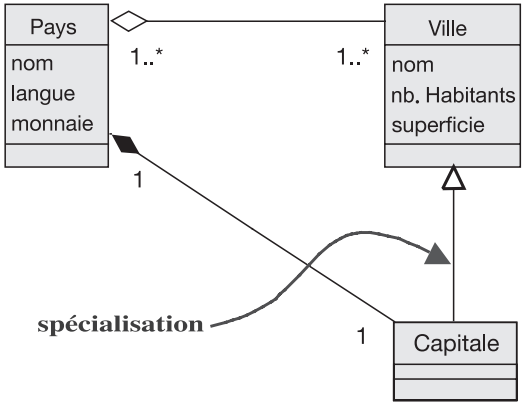
Figure 4-15.
Solution plus complète avec une contrainte



Il faut noter que nous avons considéré qu'une ville peut appartenir à plusieurs pays dans un souci de généralité, et pour nous prémunir d'éventuelles remarques sur le statut particulier passé ou futur de villes telles que Berlin ou Jérusalem… Du coup, la relation ne peut être qu'une agrégation.

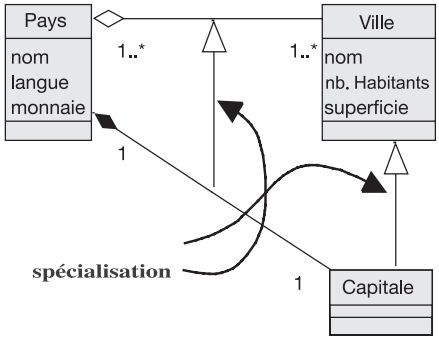
Enfin, si nous voulons préciser qu'une capitale est une ville, mais qu'elle possède des propriétés spécifiques, il faut alors en faire une sous-classe et non un rôle, comme cela est montré sur le schéma présenté ci-après.

Figure 4-16.
Solution avec super-classe concrète



La classe *Capitale* peut maintenant recevoir des attributs ou associations supplémentaires, si le besoin s'en fait sentir. Cependant, pour indiquer que la multiplicité 1 du côté *Pays* sur la composition avec *Capitale* remplace la multiplicité moins contraignante « 1..* » du côté *Pays* pour les villes, il faudrait indiquer que la composition avec *Capitale* est une spécialisation de l'agrégation avec *Ville*. UML autorise effectivement la généralisation/spécialisation entre associations.

Figure 4-17.
Solution sophistiquée avec spécialisations

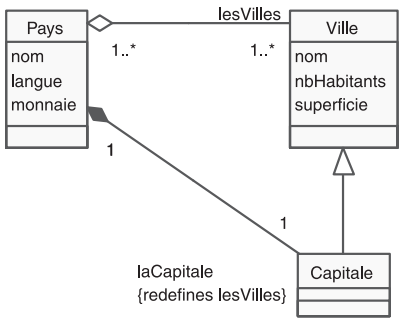


À retenir

REDÉFINITION DE PROPRIÉTÉ

UML 2 a introduit la notion de redéfinition de propriété : attribut ou terminaison d'association. Parmi les caractéristiques susceptibles d'être redéfinies, on trouve le nom, le type (qu'on peut spécialiser), la valeur par défaut, l'état de dérivation, la visibilité, la multiplicité et les contraintes sur les valeurs. Le mot-clé *redefines* remplace ainsi le symbole de spécialisation entre associations, peu implémenté par les outils du marché (voir figure 4-18).

Figure 4-18.
Solution avec redéfinition d'association (UML 2.0)



Comparez le modèle ainsi obtenu avec celui de la figure 4-11. Les deux sont corrects, « légaux » en UML et expriment à leur façon la phrase initiale. Le premier est très compact, simple à implémenter, mais très peu évolutif dans le cas où il faudrait répondre à de nouvelles demandes d'un utilisateur. Le second est nettement plus complexe à implémenter, mais très souple ; il résistera longtemps à l'évolution des besoins utilisateurs. Le choix entre les deux solutions doit donc se faire en fonction du contexte : faut-il privilégier la simplicité, les délais de réalisation, ou au contraire la pérennité et l'évolutivité ?

Il faut enfin noter que la super-classe *Ville* de la figure 4-18 n'est pas une classe abstraite. Cela est cohérent puisqu'elle ne possède qu'une seule sous-classe. En effet, l'objectif d'une classe abstraite est de factoriser des propriétés

communes à plusieurs sous-classes, et non pas à une seule ! Il est ainsi tout à fait possible qu'un modèle complet contienne une spécialisation unique, mais la super-classe ne doit alors pas être abstraite.

MODÉLISATION DU DOMAINE EN PRATIQUE

EXERCICE 4-3.
La modélisation du domaine en pratique

Modélisez l'utilisation de feutres et de stylos...

L'énoncé est volontairement vague !

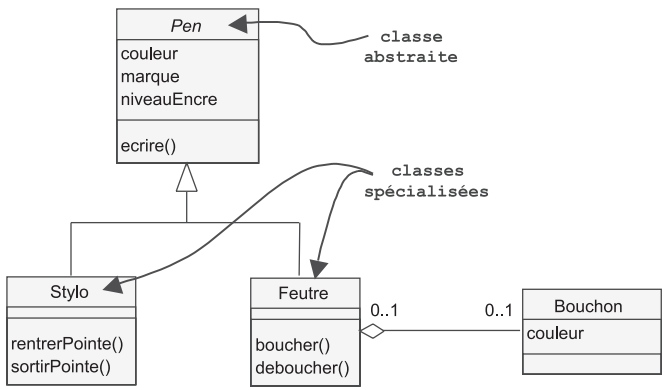
solution

Un énoncé aussi imprécis n'est pas rare au démarrage de projets réels… Nous allons cependant pouvoir construire sans difficulté un diagramme de classes pertinent en utilisant notre connaissance usuelle de ces notions de feutres et de stylos.

Nous commençons donc par identifier deux classes : *Feutre* et *Stylo*, qui ont un certain nombre de propriétés communes (couleur, marque, etc.) mais qui comptent aussi des différences (considérons par exemple que les feutres ont un bouchon, alors que les stylos n'ont qu'une pointe rétractable). Un modélisateur averti voit là immédiatement la possibilité d'une relation de généralisation/spécialisation. Il introduit donc une classe abstraite pour factoriser les caractéristiques communes.

Le modèle peut déjà ressembler au diagramme suivant :

Figure 4-19.
Première version du diagramme de classes



Prenez note des multiplicités entre *Feutre* et *Bouchon* : un feutre peut perdre son bouchon, et un bouchon le corps de son feutre d'origine. Cette notion de

Corps est intéressante, et partagée par les stylos et les feutres. Par souci d'homogénéité avec *Bouchon*, nous l'ajoutons donc.

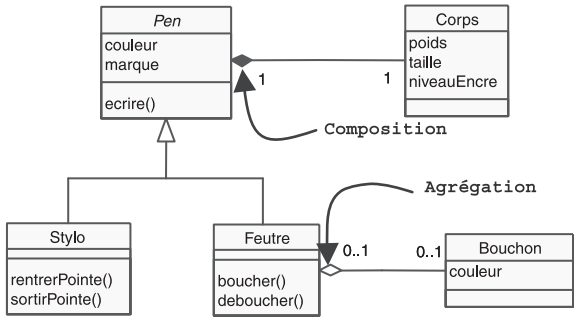


Figure 4-20.
Deuxième version du diagramme de classes

La relation entre *Pen* et *Corps* est évidemment une composition. En revanche, il n'y a pas forcément cohérence des cycles de vie entre *Feutre* et *Bouchon*, comme me le prouve tous les jours ma petite fille de 3 ans[2] ! Notez également que l'attribut *niveauEncre* a été déplacé dans la classe *Corps*. Ce déplacement d'attribut d'une classe à l'autre est habituel surtout dans le cas de relations de composition ou d'agrégation, afin que les classes soient plus homogènes et cohésives.

Pour compléter encore notre modèle, introduisons le concept de feutre avec correcteur incorporé et surtout une classe pour modéliser l'utilisateur et/ou le propriétaire du *Pen*.

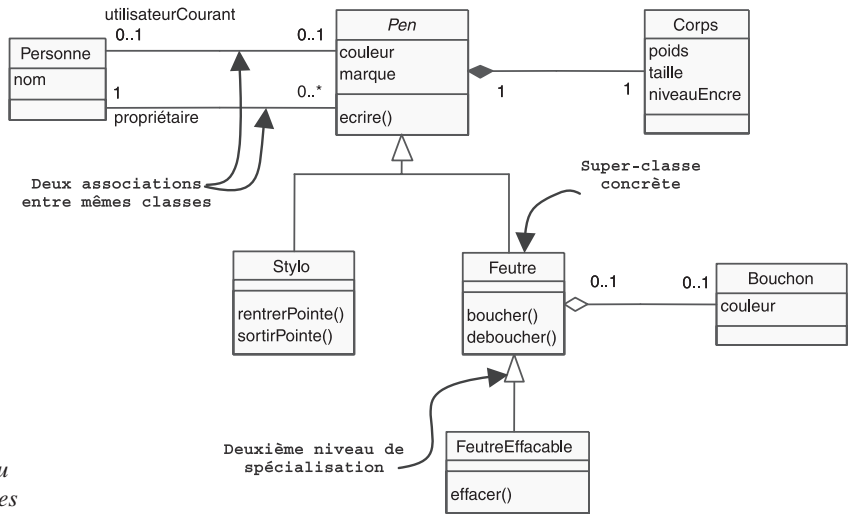


Figure 4-21.
Troisième version du diagramme de classes

Remarquez l'utilisation judicieuse qui est faite des deux associations entre les classes *Personne* et *Pen* : on distingue ainsi avec précision, grâce aux noms

2. Pour dire vrai, au fil des nouvelles éditions Noémie a grandi également. Elle a aujourd'hui 8 ans. mais elle égare encore régulièrement ses bouchons de feutres… ☺

des rôles du côté *Personne*, les multiplicités qui sont totalement différentes dans les deux cas :

- Une personne peut jouer le rôle de propriétaire par rapport à un nombre quelconque de *pens*, mais un *pen* n'a qu'un et un seul propriétaire.
- Une personne peut utiliser au maximum un *pen* à la fois, et un *pen* peut avoir au maximum un utilisateur.

La spécialisation de *Feutre* en *FeutreEffacable* est remarquable pour les raisons suivantes :

- *Feutre* n'est pas devenue une classe abstraite et ne possède qu'une seule spécialisation.
- La spécialisation ne concerne que le comportement.
- On doit donc retenir qu'une super-classe n'est pas forcément abstraite (sinon on n'aurait pas besoin de l'aide visuelle du nom en italique comme pour *Pen*), et que la relation de généralisation/spécialisation ne conduit pas toujours à un « arbre » d'héritage.

À retenir

LA CONTRAINTE {FROZEN}

Cette contrainte standard en UML[b] permet d'ajouter une information détaillée, mais qui peut être intéressante, sur un diagramme de classes :

– Pour un attribut, le fait que sa valeur ne change jamais pendant la vie d'un objet (par exemple, la marque d'un *Pen*).

– Pour une association, le fait qu'un lien entre deux objets ne puisse plus jamais être modifié après sa création (par exemple, le lien de composition entre *Pen* et *Corps*, mais pas celui entre *Feutre* et *Bouchon*).

Par défaut, les attributs et les associations ne sont pas `{frozen}`.

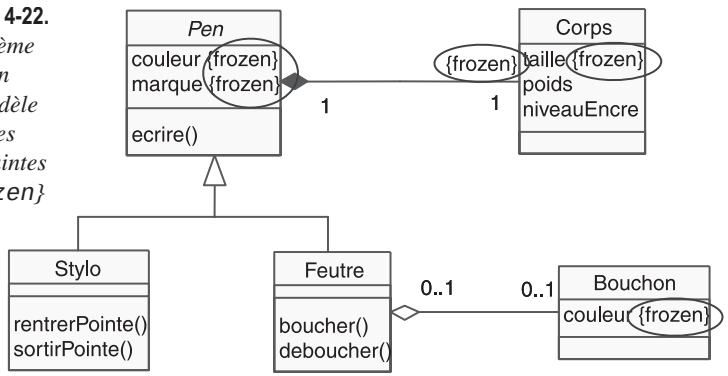


Figure 4-22. *Deuxième version du modèle avec les contraintes* `{frozen}`

b. En fait, la contrainte prédéfinie {frozen} semble avoir disparu des récents documents spécifiant UML 2. Elle existait pourtant depuis UML 1.3, comme en atteste le *UML User Guide* de G. Booch. Du fait de son intérêt, nous préconisons de continuer à l'utiliser, même si elle ne fait plus partie du standard UML.

Dans le même esprit, on aurait pu préciser que l'arbre d'héritage de *Pen* n'est pas complet (il y a certainement d'autres sortes de *Pen* que les stylos et les feutres) en lui adjoignant la contrainte prédéfinie {`incomplete`}.

EXERCICE 4-4.
La modélisation du domaine en pratique (suite)

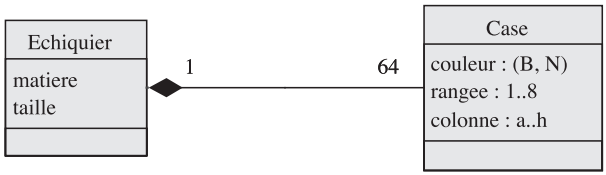
Proposez un cadre de modélisation des règles du jeu d'échecs.

Attachez-vous d'abord au matériel utilisé par les joueurs, puis à la notion de partie.

solution

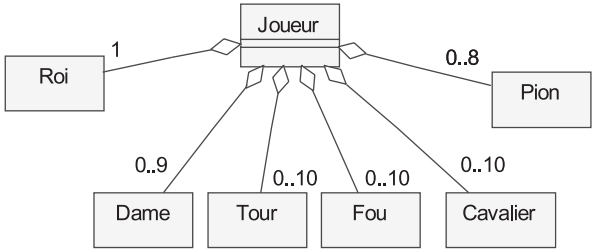
Commençons par interviewer un « expert métier » : le jeu d'échecs se joue à deux joueurs sur un échiquier carré composé de 64 cases, alternativement noires et blanches.

Figure 4-23.
Modélisation d'un échiquier et de ses cases



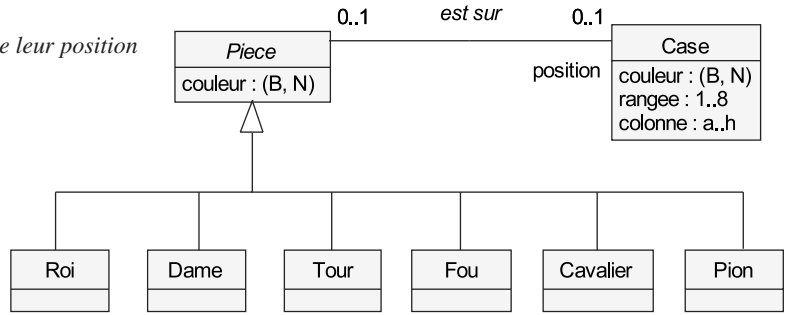
Chaque joueur possède initialement huit pions, ainsi qu'un roi, une dame, deux tours, deux fous et deux cavaliers. Par le biais de la promotion des pions en huitième rangée (transformation obligatoire en pièce à choisir sauf un roi), un joueur peut ainsi posséder jusqu'à neuf dames, dix tours, cavaliers ou fous et les multiplicités instantanées ne sont pas si évidentes !

Figure 4-24.
Modélisation des pièces de chaque joueur



Il ne peut y avoir qu'une pièce au maximum sur une case donnée. Les positions initiales des pièces sont représentées sur la figure précédente.

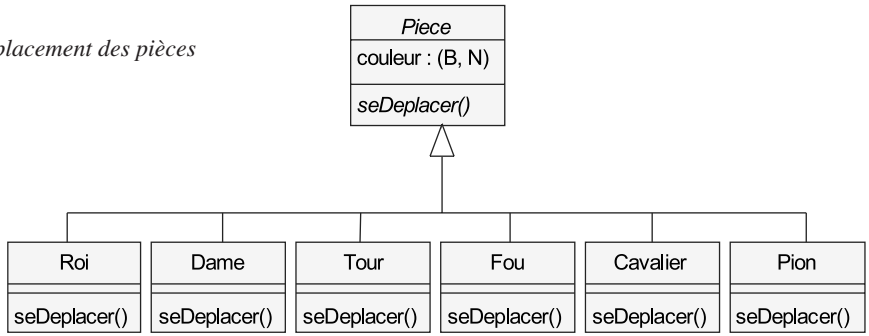
Figure 4-25.
Modélisation des pièces et de leur position



L'introduction de la classe abstraite *Piece* est tout à fait naturelle. Tout d'abord, il s'agit bien d'un mot faisant partie du vocabulaire du domaine. Ensuite, cela permet d'exprimer le concept de position par un rôle unique de l'association entre les classes *Piece* et Case. Sur une case donnée, il ne peut pas y avoir plus d'une pièce à la fois. Et une pièce est soit sur une case, soit hors du jeu (prise).

Les pièces ont chacune leur mode de déplacement propre.

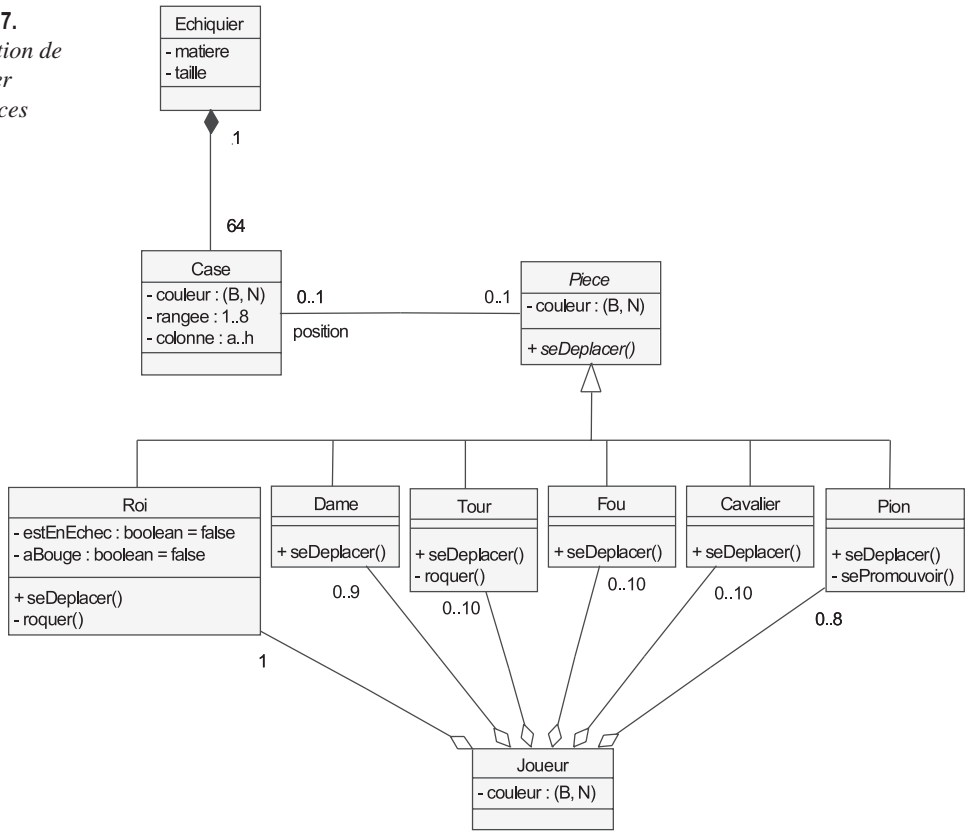
Figure 4-26.
Modélisation du déplacement des pièces



Le déplacement des pièces est typiquement polymorphe. Chaque instance se déplace en fonction de l'algorithme déclaré au niveau des sous-classes concrètes.

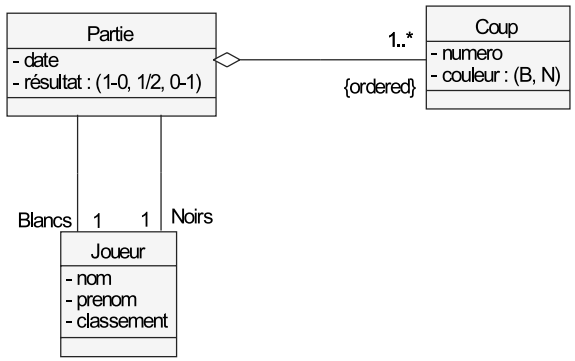
En complétant l'interview de notre expert métier, nous ajoutons quelques attributs et opérations privées pour peaufiner la première partie du modèle (voir figure 4-27).

Figure 4-27.
Modélisation de l'échiquier et des pièces



À chaque tour, un joueur bouge une pièce de son camp (blanc ou noir). On ne peut ni passer son tour, ni jouer deux fois de suite. C'est toujours les blancs qui commencent la partie. Une partie est donc une suite ordonnée de coups.

Figure 4-28.
Modélisation du déroulement de la partie



Si nous relions les concepts de coup, de pièce et de case, nous obtenons la figure 4-29.

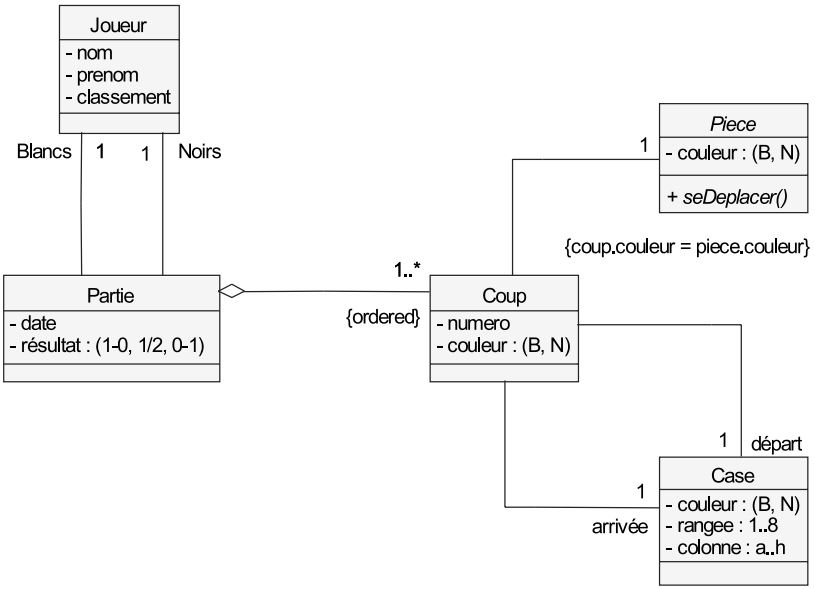


Figure 4-29.
Modélisation détaillée du déroulement de la partie

Nous arrêtons là cette exploration du jeu d'échecs par le biais de la modélisation statique, que nous pourrions encore détailler.

Par ailleurs, si nous souhaitons ajouter d'autres règles plus dynamiques concernant par exemple la fin de la partie (cas de mat, pat, abandon, partie nulle, etc.), un diagramme d'états est tout à fait approprié. Nous le présenterons au chapitre 6 (exercice 6-1).

LES CLASSES STRUCTURÉES UML 2

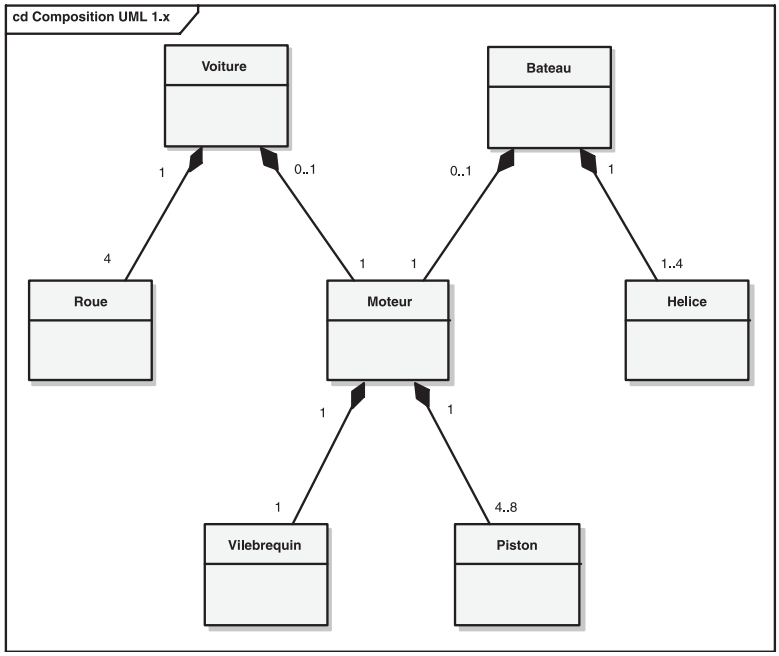
EXERCICE 4-5.
Limitations de la relation de composition

La figure 4-30[3] explique que les voitures et les bateaux possèdent un moteur, que les voitures ont des roues alors que les bateaux ont des hélices, et que les moteurs se décomposent tous de façon similaire. La composition exprime également le fait que la même instance de *Moteur* ne peut pas appartenir simultanément à une instance de *Voiture* et une instance de *Bateau*, même si la classe *Moteur* est partagée entre les classes *Voiture* et *Bateau*. C'est pour

3. Cet exercice est inspiré de l'excellent article de Conrad Bock intitulé « UML 2 Composition Model » et publié dans le « Journal of Object Technology », Vol. 3, N° 10, November-December 2004. Il est disponible sur le site web suivant : http://www.jot.fm/issues/issue_2004_11/column5.

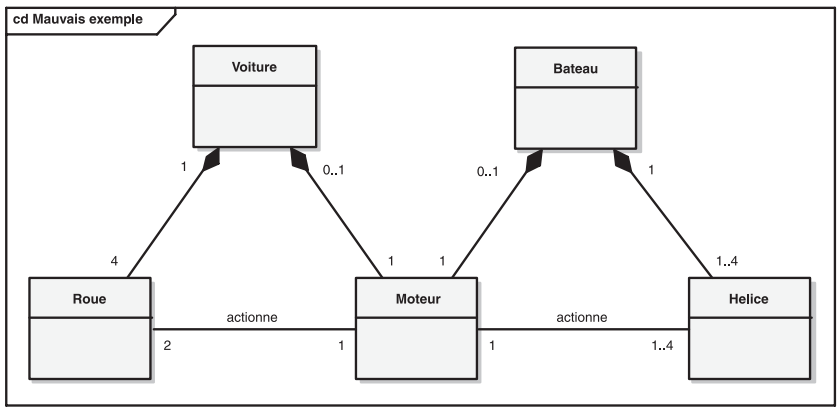
cela que les multiplicités concernées sont optionnelles (0..1)[4]. La composition signifie encore qu'une instance de *Moteur* est détruite quand son instance conteneur de *Voiture* ou de *Bateau* est détruite.

Figure 4-30.
Utilisation de la composition UML 1.x



La relation de composition telle qu'elle existait dans UML 1.x fonctionne bien pour modéliser de la décomposition hiérarchique. Cependant, elle présente des limitations significatives lorsqu'il s'agit de relier des éléments au même niveau de décomposition, comme tenté sur le schéma 4-31.

Figure 4-31.
Exemple douteux d'associations au même niveau de décomposition



4. Il aurait été encore plus précis de laisser les multiplicités à 1 exactement, mais de poser une contrainte {xor} entre les deux compositions, comme sur la figure 4-4.

Expliquez pourquoi le diagramme précédent n'est pas satisfaisant et proposez une meilleure solution en utilisant le nouveau concept UML 2 de classe structurée.

solution

Les associations ajoutées à la figure 4-31 par rapport à la 4-30 essaient d'exprimer le fait que les moteurs actionnent des roues dans les voitures et des hélices dans les bateaux. Cependant, les associations sont définies globalement pour tous les moteurs, pas dans le contexte de voitures ou de bateaux particuliers. Cela signifie donc que :

- Le moteur d'une certaine voiture peut actionner les hélices d'un bateau, ou les roues d'une autre instance de voiture.
- Chaque moteur est censé actionner à la fois des roues et des hélices. Si l'on avait mis une multiplicité 0..1 au lieu de 1 du côté *Moteur*, on aurait pu avoir cette fois-ci des moteurs n'actionnant rien du tout.
- Un moteur peut actionner les deux roues gauches d'une voiture, au lieu des roues avant. L'association ne spécifie pas quel couple de roues est actionné.

Nous voyons donc clairement que le modèle proposé n'est pas du tout satisfaisant. On pourrait dessiner un grand nombre de digrammes d'objets cohérents avec le diagramme de classes précédent, mais complètement stupides !

UML 2 traite le problème en introduisant un nouveau type de diagramme appelé « diagramme de structure composite », avec les nouveaux concepts de classe structurée, participant et connecteur.

À retenir

CLASSE STRUCTURÉE

Une classe structurée est une classe dotée d'une structure interne. Elle contient des participants (*parts*[c]) reliées par des connecteurs. Une instance d'une classe structurée contient un objet ou un ensemble d'objets correspondant à chaque participant. Au sein de son conteneur, un participant possède un type et une multiplicité. Tous les objets d'un objet structuré unique sont implicitement reliés par le fait qu'ils sont contenus par le même objet.

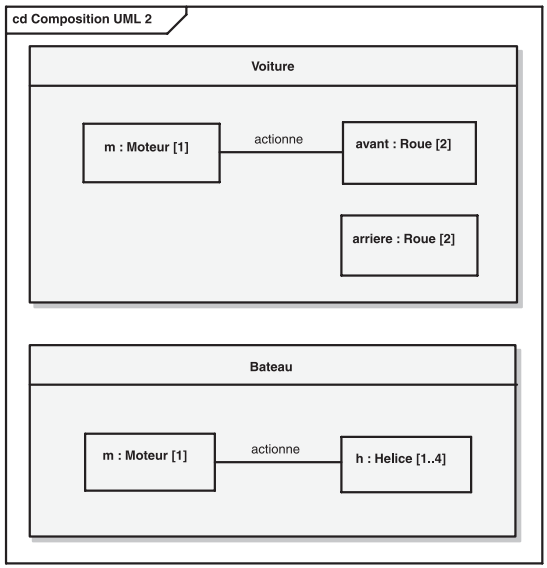
CONNECTEUR

Un connecteur est une relation contextuelle entre deux participants dans une classe structurée. La différence avec une association classique est la suivante : chaque association est indépendante des autres, alors que tous les connecteurs d'une classe structurée partagent un contexte unique.

c. Le terme anglais *part* doit plutôt être traduit par participant que partie. Il s'agit plus d'une notion de rôle joué par des éléments typés dans le contexte d'une classe que de contenance de définition.

Nous allons ainsi décrire deux contextes différents : celui de la classe *Voiture* et celui de la classe *Bateau*. À l'intérieur de chaque contexte, les rectangles représentent des participants avec pour chacune un nom de rôle et un type. Ces participants sont différents de simples instances car ils ne sont pas soulignés.

Figure 4-32.
Composition dans un contexte avec UML 2

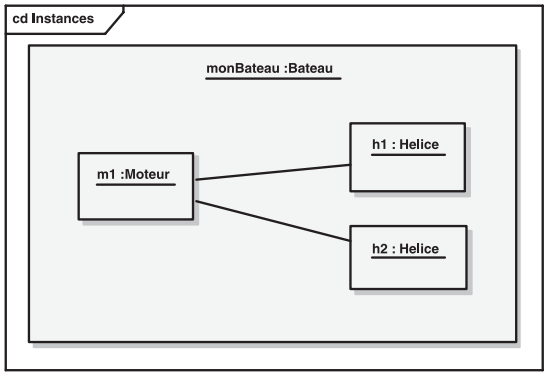


La figure précédente exprime correctement que :

- dans chaque voiture, il y aura un moteur, deux roues avant et deux roues arrière ;
- le moteur actionne les roues avant situées dans la même voiture que le moteur ;
- ce moteur n'actionne rien d'autre dans la voiture, ni dans d'autres voitures ou dans des bateaux.

Le même raisonnement s'applique pour les bateaux. Un exemple de diagramme d'objets compatible est donné ci-après. Notez le soulignement permettant de distinguer les instances par rapport aux participants.

Figure 4-33.
Diagramme d'objets UML 2



EXERCICE 4-6.
Description d'une chaîne HI-FI

À retenir

NOTION DE PORT

Un port est un point d'interaction individuel entre une classe structurée et son environnement. La classe (et ses ports) peut avoir des interfaces requises et fournies pour définir son comportement visible de l'extérieur. Lorsqu'on crée une instance de classe structurée, ses ports sont créés avec elle. À sa destruction, ses ports sont détruits. On peut également connecter des ports à des participants internes ou à la spécification de l'objet dans son ensemble.

solution

Nous avons essayé de décrire, de manière simplifiée mais réaliste, un amplituner connectable à un lecteur DVD ainsi qu'à d'autres entrées possibles. Nous avons pour cela spécifié les différents ports physiques présents sur ce type d'appareil, ainsi que l'interface homme-machine disponible sur la face avant de l'appareil[5]. Notez les symboles différents de l'interface fournie (*lollipop*) et de l'interface requise (*socket*) qui sont une nouveauté UML 2. Les interfaces sont définies en extension sur la droite du diagramme 4-34.

Nous allons maintenant connecter les éléments entre eux, sans oublier les enceintes, antennes, etc. Pour cela, nous allons utiliser le concept de collaboration avec sa définition UML 2.

À retenir

COLLABORATION

Une collaboration décrit une relation contextuelle qui prend en charge l'interaction entre un jeu de participants. Un rôle est la description d'un participant. Contrairement à une classe structurée, une collaboration ne détient pas les instances liées à ses rôles. Ces dernières existent avant l'établissement d'une instance de la collaboration, mais la collaboration les rassemble et établit des liens pour les connecter.

5. Il est clair que nous aurions pu utiliser aussi bien des composants que des classes structurées. La différence entre les deux n'est d'ailleurs pas significative en UML 2, le composant étant simplement de plus grosse granularité, mais conceptuellement équivalent (contrairement à la notion de composant UML 1.x).

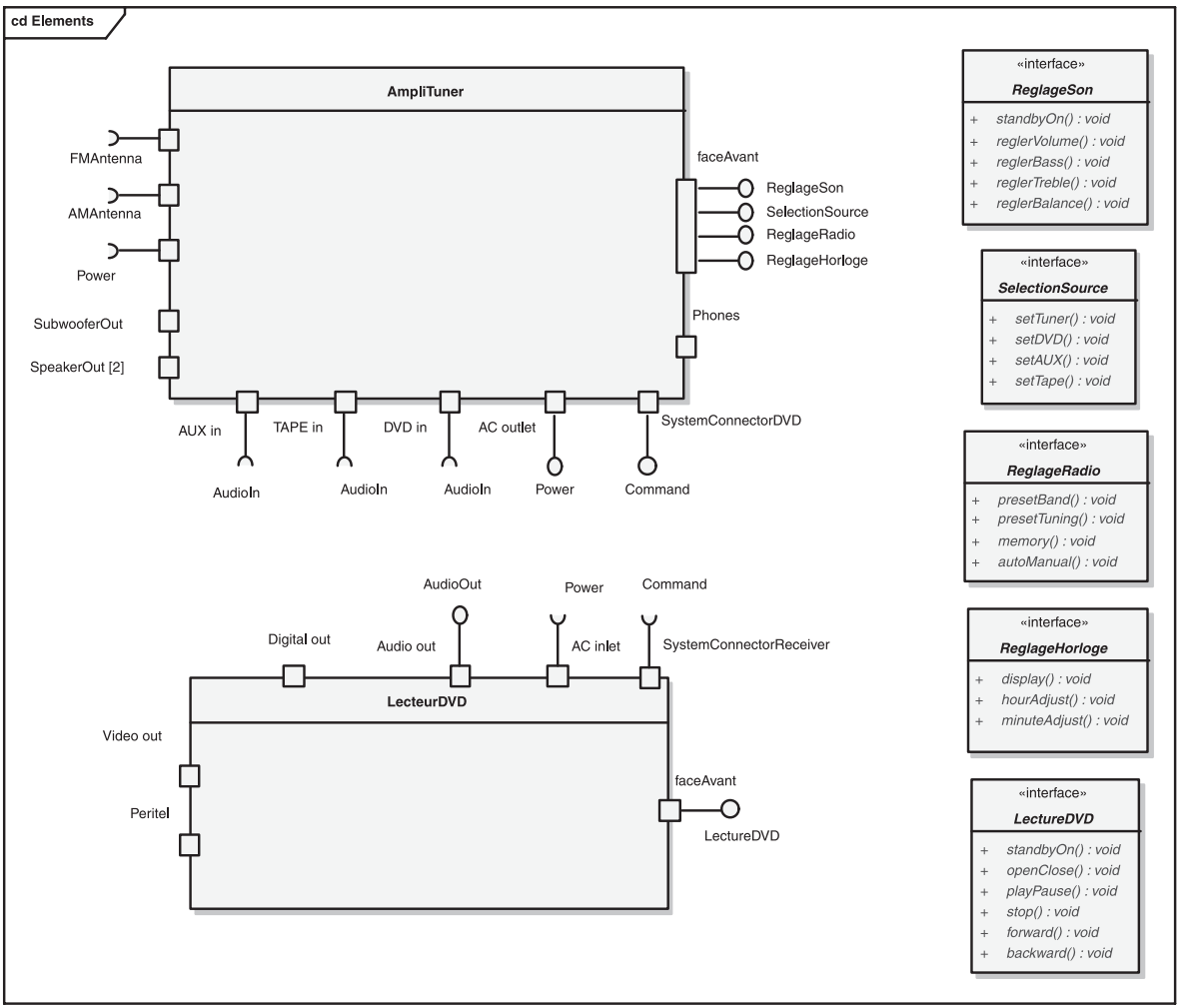


Figure 4-34.
Eléments de la chaîne HI-FI

La chaîne HI-FI va être ainsi vue comme une collaboration rassemblant un certain nombre de participants dans un objectif commun, comme représenté sur la figure suivante. Nous avons mixé volontairement les connecteurs simples et les connecteurs montrant l'imbrication des interfaces fournies et requises.

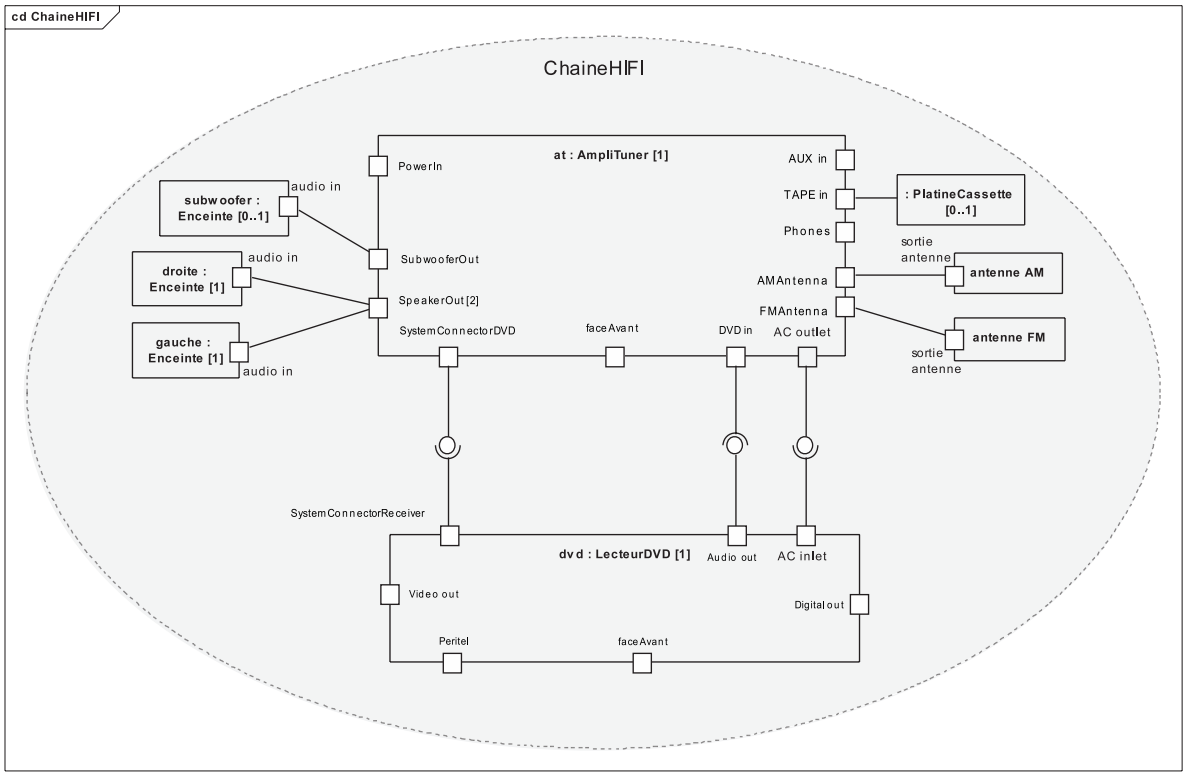


Figure 4-35.
La chaîne HI-FI vue comme une collaboration UML 2

Qu'est-ce qu'un Home Cinéma, sinon une chaîne HI-FI connectée à un dispositif de projection (abstraction pour télévision, vidéoprojecteur, etc.) ? Nous pouvons le représenter par une nouvelle collaboration faisant intervenir plus de participants de type *Enceinte*, et un participant supplémentaire de type *Dispositif Projection*.

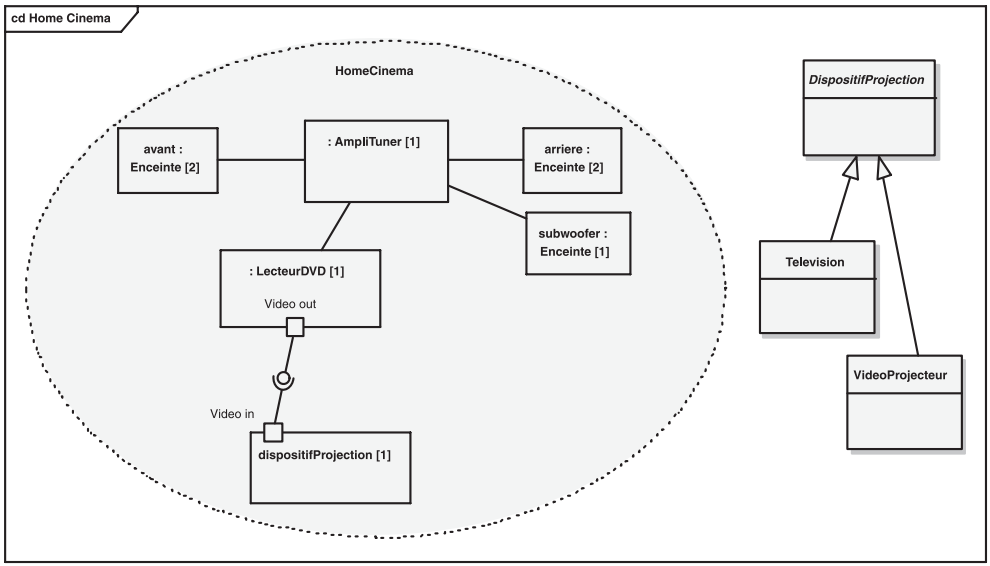


Figure 4-36.
Le « Home Cinema » vu comme une collaboration UML 2

DÉCOUVERTE D'UN « PATTERN »

EXERCICE 4-7.
Découverte guidée du pattern « Composite »

Proposez une solution élégante qui permette de modéliser le système de gestion de fichiers suivant :

1. **les fichiers, les raccourcis et les répertoires sont contenus dans des répertoires et possèdent un nom ;**
2. **un raccourci peut concerner un fichier ou un répertoire ;**
3. **au sein d'un répertoire donné, un nom ne peut identifier qu'un seul élément (fichier, sous-répertoire ou raccourci).**

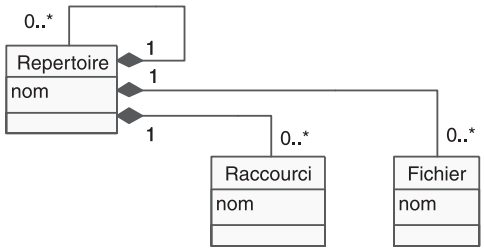
solution

Commençons par modéliser les trois phrases, une à une.

1. *Les fichiers, les raccourcis et les répertoires sont contenus dans des répertoires et possèdent un nom.*

Chacun des trois concepts doit être représenté par une classe. La contenance est modélisée par une composition, car la multiplicité du côté contenant est égale à 1, et la destruction d'un répertoire entraîne la destruction de tout ce qu'il contient.

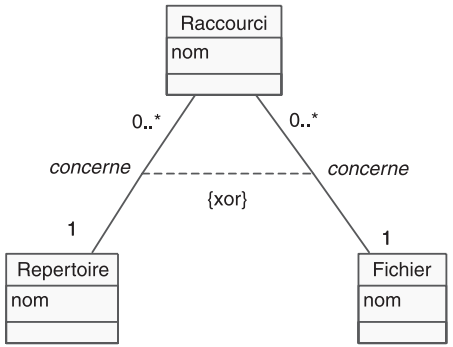
Figure 4-37.
Modélisation de la phrase 1



Deux associations en exclusion mutuelle traduisent parfaitement la deuxième phrase :

2. *Un raccourci peut concerner un fichier ou un répertoire.*

Figure 4-38.
Modélisation de la phrase 2

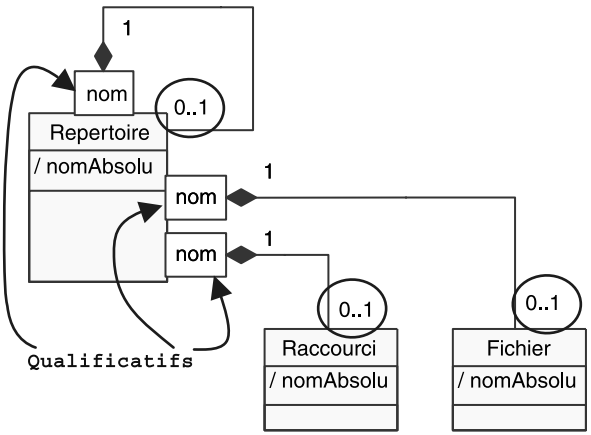


Les choses se compliquent quand il s'agit de modéliser la troisième phrase :

3. *Au sein d'un répertoire donné, un nom ne peut identifier qu'un seul élément (fichier, sous-répertoire ou raccourci).*

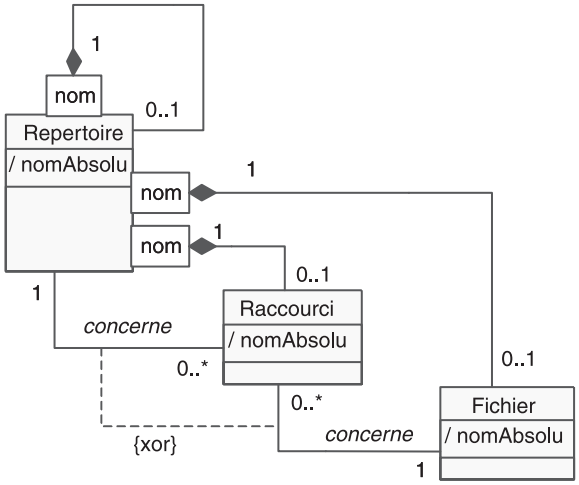
La solution la plus évidente consiste à qualifier chacune des trois compositions avec l'attribut nom. Ce qualificatif représente en fait le nom relatif de chaque élément dans son répertoire englobant. On notera la réduction de la multiplicité de l'autre côté du qualificatif. Pour modéliser le nom absolu, un attribut dérivé est tout à fait approprié, puisqu'il peut se déduire de la succession des noms relatifs.

Figure 4-39.
Modélisation de la phrase 3



Que pensez-vous du diagramme suivant, regroupant les modèles des trois phrases ?

Figure 4-40.
Première version du modèle



Le modèle obtenu semble bien répondre aux trois phrases de l'énoncé. Pourtant, il n'est pas tout à fait correct ! En effet, d'après la figure précédente, deux fichiers ou deux raccourcis ne peuvent pas avoir le même nom au sein d'un même répertoire, mais rien n'empêche en revanche qu'un fichier et un raccourci aient le même nom…

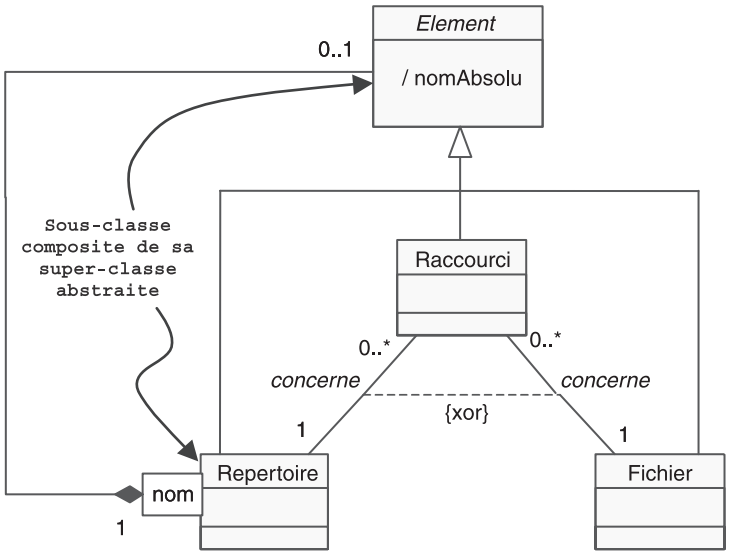
Ce léger défaut met en fait en évidence un problème majeur : il nous faut un qualificatif unique pour tout type d'élément contenu dans un répertoire et non pas un qualificatif pour chacun. Or, nous comptons trois compositions : il faut donc modifier le modèle en profondeur afin de n'avoir plus qu'une composition à qualifier. Comment faire ?

La solution est en fait contenue dans la formulation de la troisième phrase :

« Au sein d'un répertoire donné, un nom ne peut identifier qu'un seul *élément* (fichier, sous-répertoire ou raccourci). »

Le mot « élément » doit nous permettre de trouver l'idée salvatrice… Que contiennent les répertoires ? Des fichiers, des raccourcis et d'autres répertoires. Oui, mais encore, comment pouvons-nous tous les appeler ? Des éléments ! Si nous ajoutons une super-classe abstraite *Element* généralisant les fichiers, les raccourcis et les répertoires, les trois compositions se résolvent en une seule, avec un qualificatif unique, et le tour est joué ! Voici le modèle que l'on obtient alors :

Figure 4-41.
Version finale élégante



Ce qui est étonnant dans cette solution, et qui explique qu'elle ne soit pas si facile à trouver, c'est la double relation asymétrique entre les classes *Repertoire* et *Element* :

- *Repertoire* est un composite par rapport à *Element*.
- *Repertoire* est une sous-classe d'*Element*.

À retenir

LE PATTERN « COMPOSITE »

La solution de la figure suivante a été décrite d'une façon plus générale dans l'ouvrage de référence sur les *Design Patterns*[d], sous le nom de pattern « composite ».

Ce pattern fournit une solution élégante pour modéliser des structures arborescentes qui représentent des hiérarchies composant/composé. Le client peut ainsi traiter de la même façon les objets individuels (feuilles) et leurs combinaisons (composites).

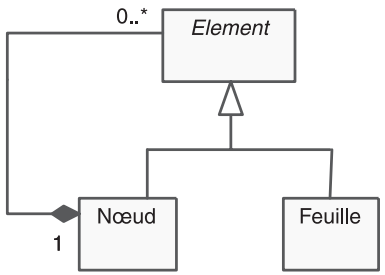
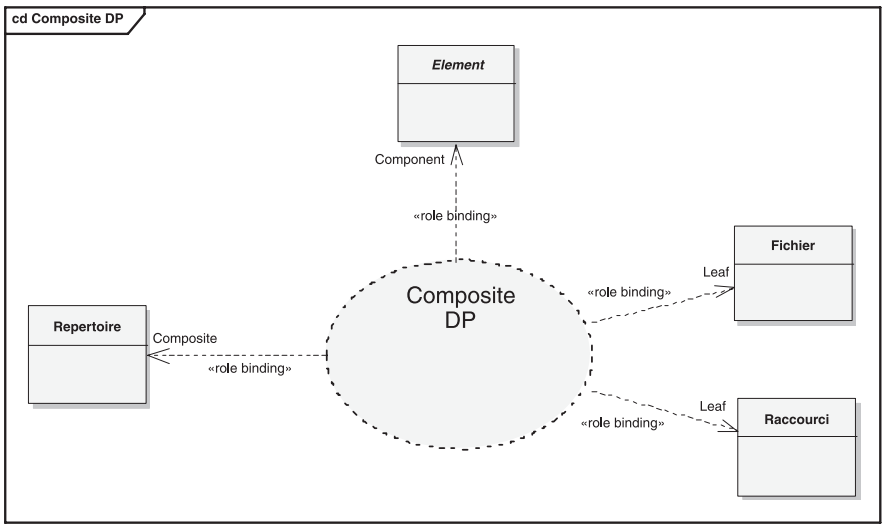


Figure 4-42.
Pattern du composite sous forme simple

d. *Design Patterns: Elements of Reusable Object-Oriented Software*, E. Gamma *et al*., 1995, Addison-Wesley.

Une autre façon d'utiliser un design pattern consiste à le représenter comme une collaboration nommée, reliée aux classes qui jouent des rôles dans la collaboration. Chaque dépendance avec le mot-clé « `role binding`» est nommée par le rôle générique des classes dans le design pattern. Il s'agit d'une manière très pratique d'appliquer un design pattern dans un contexte concret sans pour autant expliciter toutes les relations induites entre les classes qui collaborent.

Figure 4-43.
Pattern du composite appliqué sous forme de collaboration



CONSEILS MÉTHODOLOGIQUES

NOTION D'ÉTAT ET DIAGRAMME DE CLASSES

La notion d'état ne doit pas apparaître directement en tant qu'attribut sur les diagrammes de classes : elle sera modélisée dans le point de vue dynamique au moyen du diagramme d'états. Dans le diagramme de classes UML, les seuls concepts dynamiques disponibles sont les opérations.

COMMENT POSITIONNER LES OPÉRATIONS ?

En orienté objet, on considère que l'objet sur lequel on pourra réaliser un traitement doit l'avoir déclaré en tant qu'opération. Les autres objets qui posséderont une référence sur cet objet pourront alors lui envoyer un message invoquant l'opération.

OBJET OU ATTRIBUT ?

Un objet est un élément plus « important » qu'un attribut. Un bon critère à appliquer 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 l'on peut lui poser plusieurs questions, il s'agit plutôt d'un objet qui possède à son tour plusieurs attributs, ainsi que des liens avec d'autres objets.

À QUOI SERT LE DIAGRAMME D'OBJETS ?

N'hésitez pas à utiliser le diagramme d'objets pour donner un exemple, ou encore un contre-exemple, qui permette d'affiner un aspect délicat d'un diagramme de classes.

BONNE UTILISATION DE LA GÉNÉRALISATION

N'employez la relation de généralisation que lorsque la sous-classe est conforme à 100 % aux spécifications de sa super-classe.

SUPER-CLASSE ABSTRAITE OU CONCRÈTE ?

Comprenez bien pourquoi une super-classe n'est pas toujours abstraite (sinon on n'aurait pas besoin de l'aide visuelle du nom en italique), et pourquoi la relation de généralisation/spécialisation ne conduit pas toujours à un « arbre » d'héritage.

CONVENTIONS DE NOMMAGE 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.

ATTRIBUT DÉRIVÉ ET QUALIFICATIF

- Utilisez le concept d'attribut dérivé pour distinguer les attributs intéressants pour l'analyste, mais redondants car leur valeur peut être déduite à partir d'autres informations disponibles dans le modèle. Les attributs dérivés permettent à l'analyste de ne pas faire un choix de conception prématuré.
- Utilisez à bon escient les qualificatifs, sans oublier la modification de la multiplicité de l'autre côté de l'association.

AGRÉGATION OU COMPOSITION ?

Pour qu'une agrégation soit une composition, il faut vérifier les deux critères suivants :

- La multiplicité ne doit pas être supérieure à un du côté du composite.
- Le cycle de vie des parties doit dépendre de celui du composite (en particulier pour la destruction).

LIMITATIONS DE LA COMPOSITION

La relation de composition telle qu'elle existait dans UML 1.x fonctionne bien pour modéliser de la décomposition hiérarchique. Cependant, elle présente des limitations significatives lorsqu'il s'agit de relier des éléments au même niveau de décomposition. En effet, les associations sont définies globalement, pas dans le contexte de classes et encore moins d'instances particulières

Si nécessaire, vous devez avoir recours au puissant diagramme de structure composite. Celui-ci introduit les nouveaux concepts de classe structurée, participant (*part*) et connecteur. Une classe structurée est une classe dotée d'une structure interne. Elle contient des participants reliés par des connecteurs. Une instance d'une classe structurée contient un objet ou un ensemble d'objets correspondant à chaque participant. Au sein de son conteneur, un participant possède un type et une multiplicité. Tous les objets d'un objet structuré unique sont implicitement reliés par le fait qu'ils sont contenus par le même objet. Un connecteur est une relation contextuelle entre deux participants dans une classe structurée. La différence avec une association classique est la suivante : chaque

association est indépendante des autres, alors que tous les connecteurs d'une classe structurée partagent un contexte unique.

LA STRUCTURATION DU MODÈLE STATIQUE

- La structuration d'un modèle statique est une activité délicate. Elle doit s'appuyer sur deux principes fondamentaux : *cohérence* et *indépendance*. Le premier principe consiste à regrouper les classes qui sont proches d'un point de vue sémantique. À cet égard, il faut chercher l'homogénéité au niveau des critères suivants : finalité, évolution et cycle de vie. Le second principe consiste quant à lui à renforcer ce découpage initial en s'efforçant de minimiser les dépendances entre les packages.
- Le problème des associations qui traversent deux packages réside dans le fait qu'une seule d'entre elles suffit à induire une dépendance mutuelle, si elle est bidirectionnelle. Il est toutefois 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.
- Pour qu'un package soit vraiment un composant réutilisable, il ne faut pas qu'il dépende des autres packages.
- Respectez les sacro-saints principes des dépendances entre packages :
  - Pas de dépendances mutuelles.
  - Pas de dépendances circulaires.
  - Un package d'analyse contient généralement moins de 10 classes.

LA SUBJECTIVITÉ DE LA MODÉLISATION

Soyez conscient du caractère hautement subjectif de l'activité de modélisation, et du choix souvent difficile qu'il faut faire entre simplicité et évolutivité. Un modèle très compact, simple à implémenter, sera peu évolutif lorsque de nouvelles demandes émaneront des utilisateurs. Un modèle nettement plus complexe à implémenter, mais très souple, résistera mieux à l'évolution des besoins utilisateurs. Le choix entre les deux solutions doit donc se faire en fonction du contexte : faut-il privilégier la simplicité, les délais de réalisation, ou au contraire la pérennité et l'évolutivité ?

FORTE COHÉSION ET MÉTACLASSE

- Veillez à ce que vos classes n'aient pas trop de responsabilités différentes, sous peine de violer un principe fort de la conception orientée objet, appelé *forte cohésion*.
- Si vous identifiez une classe *XX* qui possède trop de responsabilités, et dont certaines ne sont pas propres à chaque instance, pensez au *pattern de la métaclasse*.

Ajoutez une classe *TypeXX*, répartissez les propriétés sur les deux classes et reliez-les par une association « * - 1 ». La classe *TypeXX* est qualifiée de « métaclasse », car elle contient des informations qui décrivent la classe *XX*. On parle aussi de type et d'exemplaires.

ÉTUDIEZ LES PATTERNS !

Apprenez à identifier les moments opportuns où il convient d'utiliser un pattern de modélisation. Contraignez-vous à les étudier sérieusement afin de ne pas « réinventer la roue » à chaque nouveau modèle !