Introduction aux objets

OURS BLANC DES CARPATHES™

ISIMA

ECOLE DOCTORALE SCIENCES FONDAMENTALES

1999

Table des matières

Table des illustrations

1. Présentation de la notion d'objet

Un *objet* est une entité cohérente rassemblant des données et du code travaillant sur ses données. Une *classe* peut être considérée comme un moule à partir duquel on peut créer des objets. La notion de *classe* peut alors être considérée comme une expression de la notion de classe d'équivalence chère aux mathématiciens. En fait, on considère plus souvent que les classes sont les *descriptions* des objets (on dit que les classes sont la *méta donnée* des objets), lesquels sont des *instances* de leur classe.

Pourquoi ce vocabulaire ? une classe décrit la structure interne d'un objet : les données qu'il regroupe, les actions qu'il est capable d'assurer sur ses données. Un objet est un état de sa classe. Considérons par exemple la modélisation d'un véhicule telle que présentée par la figure suivante :

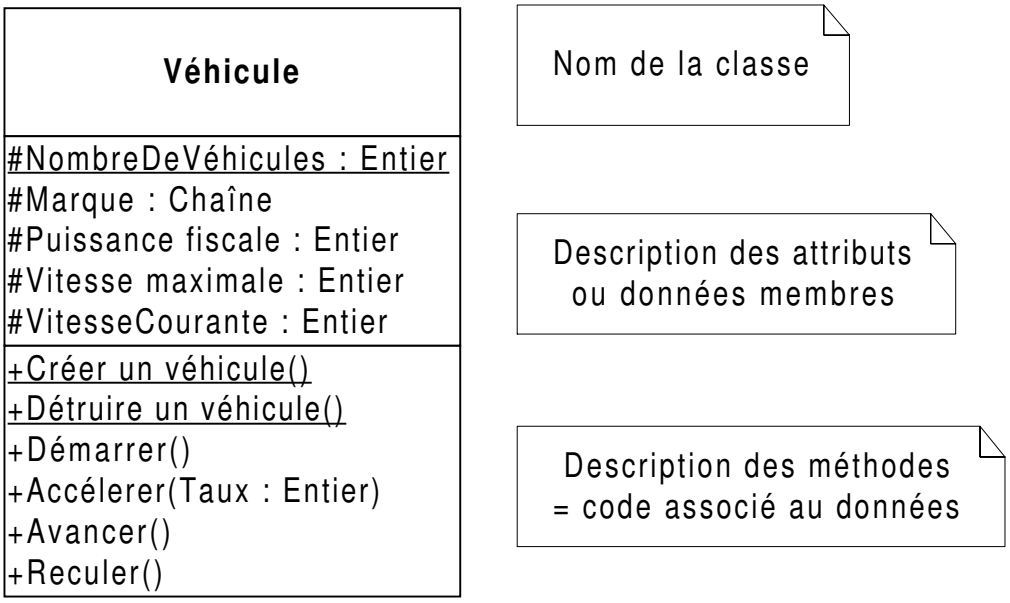

Figure 1.1 Représentation de la classe Véhicule

Dans ce modèle, un véhicule est représenté par une chaîne de caractères (sa *marque*) et trois entiers : la *puissance fiscale*, la *vitesse maximale* et la *vitesse courante*. Toutes ces données sont représentatives d'un véhicule particulier, autrement dit, chaque objet véhicule aura sa propre copie de ses données : on parle alors d'attribut *d'instance*. L'opération *d'instanciation* qui permet de créer un objet à partir d'une classe consiste précisément à fournir des valeurs particulières pour chacun des attributs d'instance.

Le schéma précédent nous permet de présenter UML (Unified Modelling language), langage de représentation des systèmes objet quasi universellement adopté de nos jours. La présente introduction ne permettra de voir qu'une infime partie d'UML au travers de son schéma de représentation statique dont le but est de présenter les différentes classes intervenant dans un modèle,

accompagnées de leurs principales relations. Nous voyons donc qu'une classe est représentée sous forme d'un rectangle lui même divisé en trois volets dans le sens de la hauteur.

❍ Le volet supérieur indique le nom de la classe

❍ Le volet intermédiaire présente les attributs et leur type sous la forme :

```
identificateurAttribut : identificateurType.
```

❍ Le volet inférieur présente les méthodes accompagnées de leur type de retour et de leurs paramètres

Le soulignement indique un membre de classe, que ce soit un attribut ou une méthode.

Finalement, les rectangles avec un coin replié sont associés aux notes ou commentaires explicatifs.

En revanche, considérons l'attribut Nombre de véhicules chargé de compter le nombre de véhicules présents à un moment donné dans la classe. Il est incrémenté par l'opération Créer un véhicule et décrémenté par l'opération Détruire un véhicule. C'est un exemple typique d'attribut partagé par l'ensemble des objets d'une même classe. Il est donc inutile et même dangereux (penser aux opérations de mise à jour) que chaque objet possède sa copie propre de cet attribut, il vaut mieux qu'ils partagent une copie unique située au niveau de la classe. On parle donc d'attribut de classe.

La figure suivante montre l'opération d'instanciation de la classe en 2 objets différents :

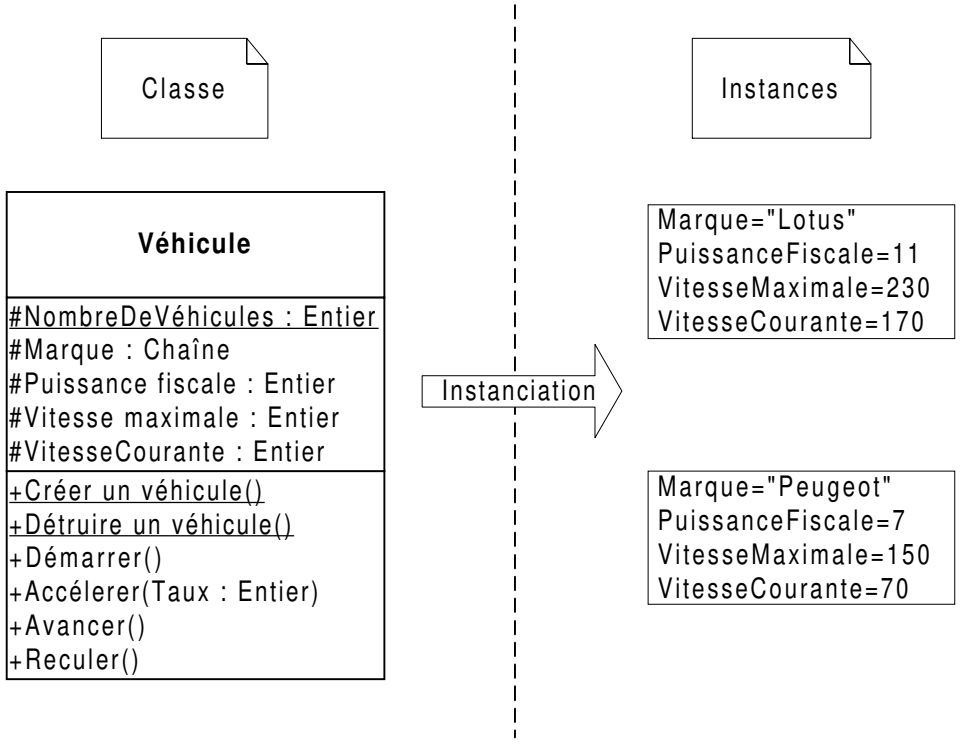

Figure 1.2 Instanciation d'une classe en deux objets

Le même raisonnement s'applique directement aux méthodes. En effet, de la même manière que nous avions établi une distinction entre attributs d'instance et attributs de classe, nous allons différencier méthodes d'instances et méthodes de classe.

Prenons par exemple la méthode Démarrer. Il est clair qu'elle peut s'appliquer individuellement à chaque véhicule pris comme entité séparée. En outre, cette méthode va clairement utiliser les attributs d'instance de l'objet auquel elle va s'appliquer c'est donc une méthode d'instance.

Considérons désormais le cas de la méthode Créer un véhicule. Son but est de créer un nouveau véhicule, lequel aura, dans un second temps, le loisir de positionner des valeurs initiales dans chacun de ces attributs d'instance. Si nous considérons en détail le processus permettant de créer un objet, nous nous apercevons que la première étape consiste à allouer de la mémoire pour le nouvel objet. Hors cette étape n'est clairement pas du ressort d'un objet : seule la classe possède suffisamment d'informations pour la mener à bien : la création d'un objet est donc une méthode de classe. Notons également qu'au cours de cette étape, l'objet recevra des indications additionnelles, telles que, par exemple, une information lui indiquant à quelle classe il appartient. En revanche, considérons la phase d'initialisation des attributs. Celle-ci s'applique à un objet bien précis : celui en cours de création. L'initialisation des attributs est donc une méthode d'instance.

Nous aboutissons finalement au constat suivant : la création d'un nouvel objet est constituée de deux phases :

- Une phase du ressort de la classe : allouer de la mémoire pour le nouvel objet et lui fournir un contexte d'exécution minimaliste
- Une phase du ressort de l'objet : initialiser ses attributs d'instance

Si ces deux phases sont clairement séparées dans un langage tel que l'objective C, elles ne le sont plus en C++ ou en Java qui agglomèrent toute l'opération dans une méthode spéciale, ni vraiment méthode d'instance, ni méthode de classe : le constructeur.

2. Le principe d'encapsulation

Sans le savoir, vous avez déjà mis le doigt sur l'un des trois grands principes du paradigme objet : l'encapsulation. Ce principe, digne héritier des principes d'abstraction de données et d'abstraction procédurale prône les idées suivantes :

- Un objet rassemble en lui même ses données (les attributs) et le code capable d'agir dessus (les méthodes)
- Abstraction de données : la structure d'un objet n'est pas visible de l'extérieur, son interface est constituée de messages invocables par un utilisateur. La réception d'un message déclenche l'exécution de méthodes.
- Abstraction procédurale : Du point de vue de l'extérieur (c'est-à-dire en fait du client de l'objet), l'invocation d'un message est une opération atomique. L'utilisateur n'a aucun élément d'information sur la mécanique interne mise

en œuvre. Par exemple, il ne sait pas si le traitement requis a demandé l'intervention de plusieurs méthodes ou même la création d'objets temporaires etc.

Dans les versions canoniques du paradigme objet, les services d'un objet ne sont invocables qu'au travers de messages, lesquels sont individuellement composés de :

- Un nom
- Une liste de paramètres en entrée
- Une liste de paramètres en sortie

La liste des messages auxquels est capable de répondre un objet constitue son interface : c'est la partie publique d'un objet. Tout ce qui concerne son implémentation doit rester caché à l'utilisateur final : c'est la partie privée de l'objet. Bien entendu, tous les objets d'une même classe présentent la même interface, en revanche deux objets appartenant à des classes différentes peuvent présenter la même interface. D'un certain point de vue, l'interface peut être vue comme un attribut de classe particulier.

En pratique, dans la plupart des langages orientés objet modernes, l'interface représente la liste des méthodes accessibles à l'utilisateur.

Il est très important de cacher les détails les détails d'implémentation des objets à l'utilisateur. En effet, cela permet de modifier, par exemple la structure de données interne d'une classe (remplacer un tableau par une liste chaînée) sans pour autant entraîner de modifications dans le code de l'utilisateur, l'interface n'étant pas atteinte. Autre exemple : considérons une classe modélisant un point en 2 dimensions pour laquelle on fournit deux méthodes renvoyant respectivement l'abscisse et l'ordonnée du point. Peu importe à l'utilisateur de savoir que le point est stocké réellement sous forme cartésienne ou si le concepteur a opté pour la représentation polaire !

Tous les langages orientés objet n'imposent pas de respecter le principe d'encapsulation. C'est donc au programmeur de veiller personnellement au grain.

3. L'héritage

L'héritage est le second des trois principes fondamentaux du paradigme orienté objet. Il est chargé de traduire le principe naturel de Généralisation / Spécialisation.

En effet, la plupart des systèmes réels se prêtent à merveille à une classification hiérarchique des éléments qui les composent. La première idée à ce sujet est liée à l'entomologie et aux techniques de classification des insectes en fonction de divers critères. Expliquons nous d'abord sur le terme d'héritage. Il est basé sur l'idée qu'un objet spécialisé bénéficie ou hérite des caractéristiques de l'objet le plus général auquel il rajoute ses éléments propres.

En terme de concepts objets cela se traduit de la manière suivante :

- On associe une classe au concept le plus général, nous l'appellerons *classe de base* ou *classe mère* ou *super - classe*.

- Pour chaque concept spécialisé, on dérive une classe du concept de base. La nouvelle classe est dite *classe dérivée* ou *classe fille* ou *sous-classe*

L'héritage dénotant une relation de généralisation / spécialisation, on peut traduire toute relation d'héritage par la phrase :

*« La classe dérivée **est une** version spécialisée de sa classe de base »*

On parle également de relation *est-un* pour traduire le principe de généralisation / spécialisation.

Nous prendrons deux exemples classiques :

3.1 Exemple 1 : les objets graphiques

Considérons un ensemble d'objets graphiques. Chaque objet graphique peut être considéré relativement à un point de base que nous représenterons par ses coordonnées cartésiennes X et Y. On lui associe également sa Couleur ainsi que l'épaisseur du trait. Hormis la création et la destruction d'objets, nous associons les méthodes suivantes à notre objet graphique :

- accès en lecture et en écriture des attributs
- affichage
- effacement
- déplacement d'un objet.

Nous obtenons alors la représentation de la classe `ObjetGraphique` présentée sur la figure de la page suivante.

Rajoutons deux classes spécialisées : `Ligne` et `Cercle`. Chacune d'entre elles rajoute ses attributs propres : le rayon pour le cercle, la longueur et l'angle pour la ligne. Ainsi, les méthodes Ligne et Cercle disposent de leurs attributs propres qui traduisent leur spécialisation et des attributs de la classe de base qui sont hérités.

Les méthodes de la classe de base sont également héritées. Les classes `Ligne` et `Cercle` n'ont pas, par exemple, à fournir de code pour la méthode `getX` chargée de renvoyer la valeur de l'attribut `X`. En revanche, elles sont libres de rajouter les méthodes qui leur sont propres, par exemple, des méthodes pour accéder aux attributs supplémentaires.

En outre, si vous regardez attentivement le schéma, vous verrez également que `Ligne` aussi bien que `Cercle` *redéfinissent* les méthodes `Afficher` et `Effacer` : en effet, un cercle ne s'affiche pas de la même manière qu'une ligne : c'est le *polymorphisme* appliqué aux méthodes `Afficher` et `Effacer` dans le cadre des classes Ligne et Cercle.

> Nous reviendrons plus en détails sur le principe de polymorphisme, pour l'heure il vous suffit de savoir qu'une méthode (ou une procédure / fonction) peut prendre différentes formes.

- Une forme faible : la surcharge qui permet d'utiliser le même nom de méthode / procédure / fonction avec des listes paramètres différentes.
- La forme forte du polymorphisme qui concerne la redéfinition d'une méthode d'une classe mère par ses classes dérivées en utilisant la même signature (même liste de paramètres et même type de retour).

Il est important de noter que les signatures (liste de paramètre de type de retour) des méthodes `Afficher` et `Effacer` sont les mêmes aussi bien dans la classe de base que dans les deux classes dérivées. Cela permet d'appeler la méthode de la même manière sur n'importe quel objet de la hiérarchie sans même savoir à quelle classe il appartient réellement ! la puissance du polymorphisme est sans borne ☺.

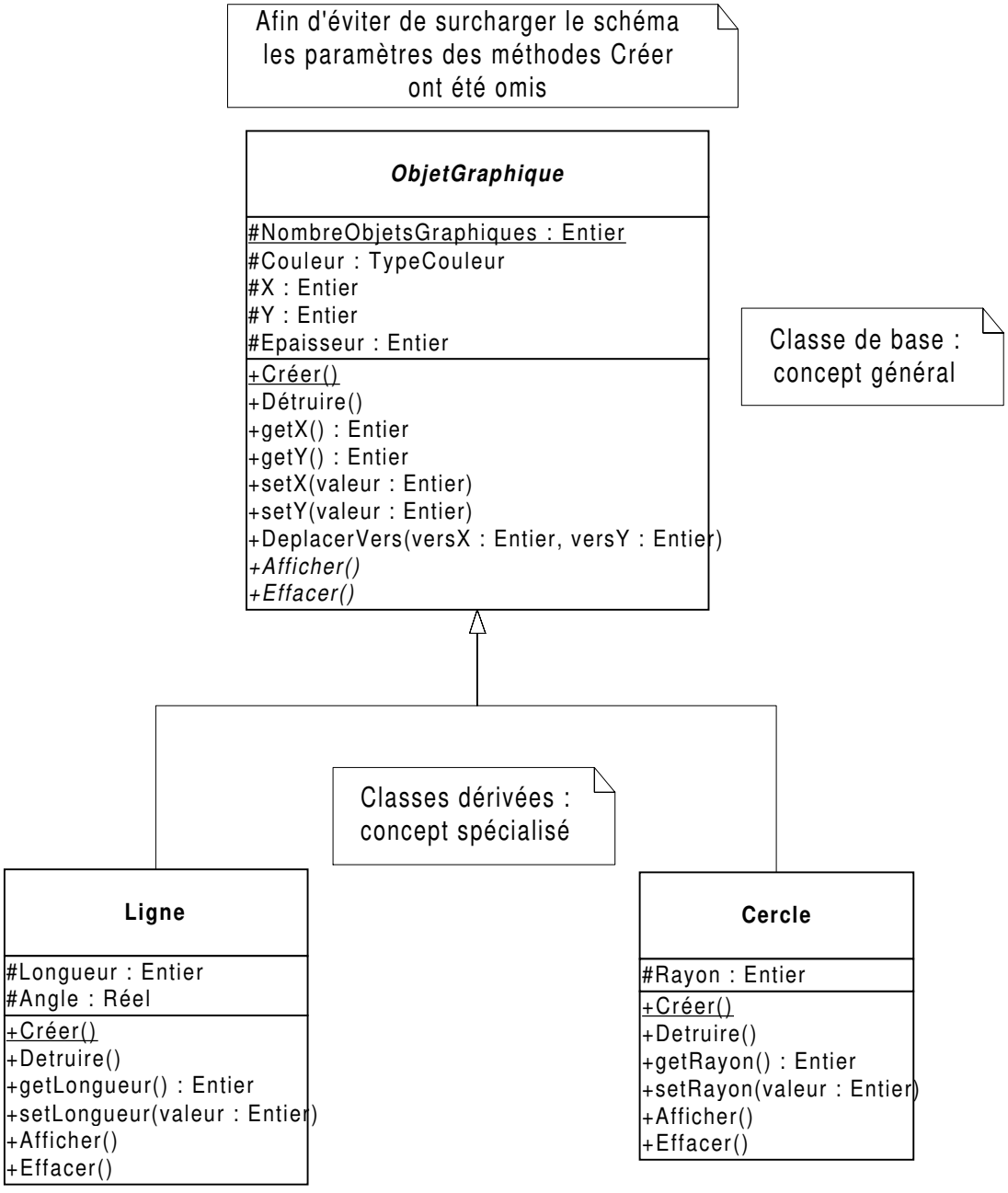

Figure 3.1 La hiérarchie de la classe ObjetGraphique

En notation UML, la relation d'héritage est signalée par une flèche à l'extrémité triangulaire et dont la pointe est orientée vers la classe mère.

Les méthodes dont le nom est en *italique* sont abstraites. On en déduit immédiatement que les classes dont le nom est italique sont également abstraites et ne peuvent donc pas directement produire d'instances.

Rappelons également que les méthodes ou les attributs soulignés sont des membres de classe.

J'attire votre attention sur le fait que la méthode `DéplacerVers` n'est pas redéfinie. En effet, de manière très littéraire, on peut l'implémenter comme suit :

```
Méthode ObjetGraphique::DéplacerVers(positionX : Entier, positionY :
Entier)
{
  [objet Effacer]
  [objet setX : positionX]
  [objet setY : positionY]
  [objet Afficher]
}
```

Programme 3.1 code générique de la méthode `DéplacerVers`

Il est aisé de voir que cette implémentation est tout à fait correcte du moment que Effacer fait bien appel à la méthode Effacer de Ligne lorsque l'on appelle la méthode `DéplacerVers` sur un objet de classe `Ligne` et réciproquement pour la classe `Cercle`. Une fois de plus, nous utilisons ici la notion de polymorphisme. En effet, appliquée à un objet de classe Cercle, la méthode `DéplacerVers` appellera `Effacer` et `Afficher` de Cercle (on notera `Cercle::Afficher` et `Cercle::Effacer`) ; appliquée à un objet de classe `Ligne`, ce sont les méthodes `Ligne::Afficher` et `Ligne::Effacer` qui seront invoquées.

Cet exemple, aussi simple, soit-il illustre bien les avantages que l'on retire à utiliser de l'héritage :

- Le code est de taille plus faible car l'on factorise au niveau des classes généralisées les comportements communs à toute une hiérarchie
- Au niveau des classes dérivées, seul le code spécifique reste à écrire, il est donc inutile de réinventer la roue à chaque étape, et le développement est plus rapide
- Modélisation d'une notion très naturelle permettant de créer des systèmes conceptuellement bien conçus.
- Le mécanisme de polymorphisme fort repose largement sur l'héritage comme nous pourrons le voir par la suite.
- Le code des classes les plus élevées dans la hiérarchie (les plus généralistes) est utilisé très souvent et gagne rapidement en fiabilité car il est très souvent sollicité et donc débogué rapidement.

- Si la hiérarchie est bien pensée, il est aisé de rajoute de nouvelles classes en considérant les différences de la nouvelle classe par rapport à celles déjà présentes dans la hiérarchie : on parle de *programmation différentielle*.

3.2 Exemple 2 modélisation d'un parc de véhicules

Cet exemple va nous montrer comme utiliser le principe de généralisation / spécialisation pour construire un système orienté objet fiable.

L'entreprise *A* possède un parc de véhicules regroupant :

- des voitures
- des camions
- des hélicoptères
- des bateaux

Elle souhaite disposer d'un modèle permettant de modéliser le fonctionnement de son parc. Le but est de créer un système de classes permettant de factoriser le plus possible les fonctionnalités de chaque type de véhicule. Partant des catégories à modéliser, il apparaît assez évident de dériver les classes `Voiture` et `Camion` d'une même classe `VéhiculeRoulant` et de laisser de côté les classes `Hélicoptère` et les `Bateau`. En outre, bien que différents, tous les véhicules partagent certaines caractéristiques de par leur caractère mobile. Ainsi des actions telles que `Démarrer`, `Accélérer`, `Ralentir` ou `Arrêter` possèdent un sens pour chaque type de véhicule. Aussi, toutes nos classes vont partager un ancêtre commun que nous appellerons `Véhicule`.

Nous obtenons alors le modèle suivant :

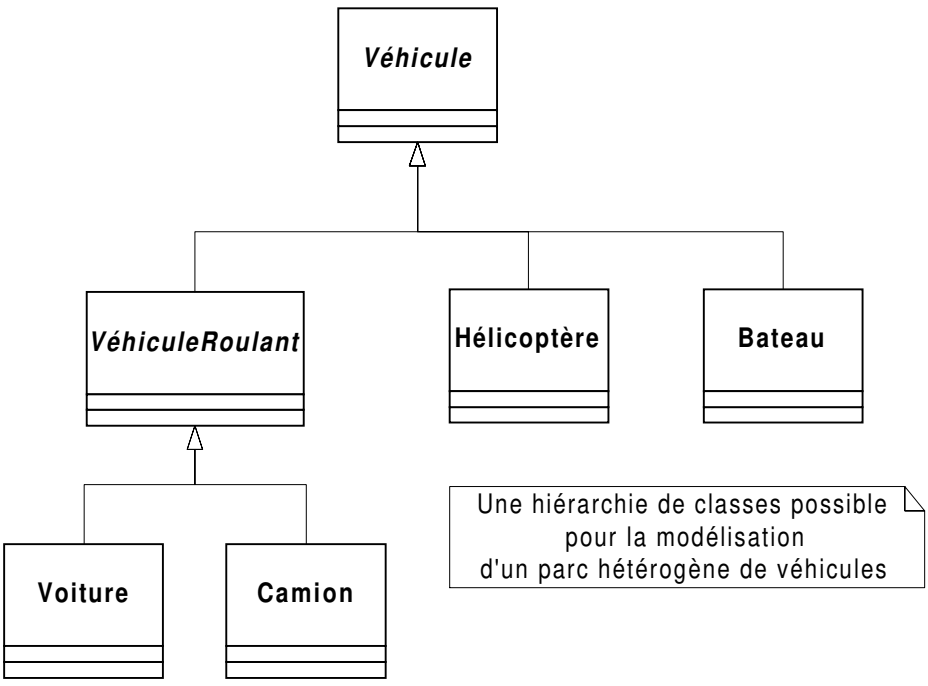

Figure 3.2 Modèle d'un parc de véhicules

Pour réaliser ce modèle, nous avons procédé par généralisation : à partir de l'ensemble des objets terminaux, nous avons cherché à établir des éléments communs permettant de mettre en évidence des classes de généralisation. Ce procédé, dit de Généralisation, est typique de la construction d'une hiérarchie de classes *ex-nihilo*.

Le procédé de Spécialisation est lui plus utilisé lorsqu'il s'agit de greffer de nouvelles classes dans un système existant. En effet, si la compagnie fait l'acquisition de quelques avions, il sera toujours possible de créer (par Généralisation) une classe `VéhiculeAérien` dont dériveront `Avion` et `Hélicoptère`. On obtient alors le modèle de la Figure 3.3.

Ce genre de classification prend parfois le nom de taxinomie, pour rendre hommage à ces précurseurs : les entomologistes qui cherchaient un moyen cohérent et pratique de classification des divers insectes qu'ils étudiaient.

Le principe de généralisation / spécialisation est particulièrement intuitif et puissant car il permet d'identifier des comportements similaires le long d'un arbre de dérivation, chaque sous classe étant à même de choisir entre redéfinir ou hériter du comportement de sa classe mère.

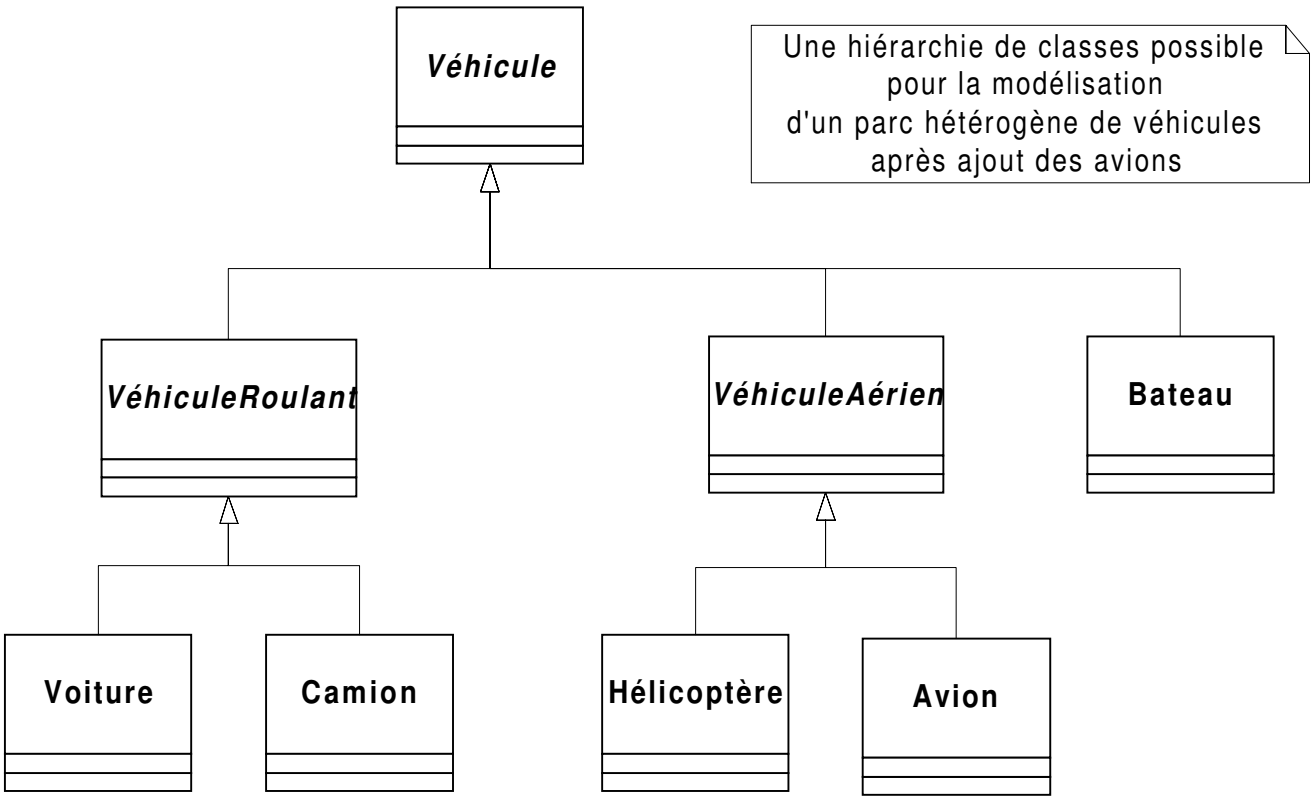

Figure 3.3 Modèle du même parc de véhicules après ajout des avions parmi les objets à modéliser

3.3 Les classes abstraites

En lecteurs attentifs, vous aurez sans doute remarqué que les titres des classes `Véhicule`, `VéhiculeRoulant` et `VéhiculeAérien` sont en italique dans la figure précédente. En sus de rendre le schéma agréable à l'œil, cette présentation n'a rien d'innocent : cela signifie que ces classes sont *abstraites*.

Une classe est dite abstraite si elle ne fournit pas d'implémentation pour certaines de ces méthodes qui sont dites méthodes abstraites. Bien entendu, étant incomplète, une classe abstraite ne peut avoir d'instance. Il appartient donc à ces classes dérivées de définir du code pour chacune des méthodes abstraites. On parlera alors de classes *concrètes* ; les classes concrètes étant les seules à même d'être instanciées.

Quel est le but des classes abstraites : définir un cadre de travail pour les classes dérivées en proposant un ensemble de méthodes que l'on retrouvera tout au long de l'arborescence. Ce mécanisme est fondamental pour la mise en place du polymorphisme. En outre, si l'on considère la classe Véhicule, il est tout à fait naturel qu'elle ne puisse avoir d'instance : un véhicule ne correspond à aucun objet concret mais plutôt au concept d'un objet capable de démarrer, ralentir, accélérer ou s'arrêter, que ce soit une voiture, un camion ou un avion.

Déterminer si une classe sera abstraite ou concrète dépend souvent du cahier des charges du logiciel. Au travers de l'étude de celui-ci, il est possible de déterminer quelles sont les classes concrètes : ce sont celles qui possèdent des instances. A partir de ces dernières, il est possible, par une démarche de généralisation, de trouver les classes abstraites qui permettront de bénéficier du polymorphisme.

3.4 Les difficultés inhérentes à l'utilisation de l'héritage

Nous venons de voir deux modèles où la décomposition était triviale. Il est souvent beaucoup plus difficile de déterminer la bonne hiérarchie. La règle est simple, comme nous l'exprimions ci-dessus, une relation d'héritage doit pouvoir se traduire par "la classe dérivée est une forme spécialisée de sa classe de base".

3.4.1 Hiérarchies trop lourdes

Il faut faire attention à ne pas trop alourdir la hiérarchie en dérivant à tour de bras. Considérons par exemple une classe `Animal` de laquelle on dérive deux classes `Chien` et `Chat`. Jusqu'ici rien à dire, on peut en effet comprendre que les différences significatives de comportement des deux races d'animaux justifient l'existence de deux classes différentes. En revanche, il serait maladroit de dériver de `Chien` des classes telles que `ChienJaune` ou `ChienNoir` sous prétexte que la couleur du pelage est différente. En effet, je ne pense pas que la couleur d'un chien soit discriminante quand à son comportement : il s'agit de toute évidence d'une simple différence de valeur d'un attribut correspondant à la couleur.

De la même manière, il faut veiller à ne pas insérer trop de classes intermédiaires. La figure de la page suivante illustre un cas typique où une classe intermédiaire ne possédant aucune dérivation propre peut être supprimée.

En effet, si la classe `Base` dispose de deux classes filles, la classe `Intermédiaire1` ne fait qu'alourdir inutilement la hiérarchie. En effet, elle est abstraite (comme en témoigne son nom en italique) donc elle ne peut pas avoir d'instances et elle n'est dérivée qu'une seule fois. On pourrait donc la supprimer pour intégrer ces caractéristiques dans la classe `Intermédiaire2`. Cet exemple permet en outre d'ajouter une touche de vocabulaire : on appelle Feuille une classe qui n'a pas de dérivées.

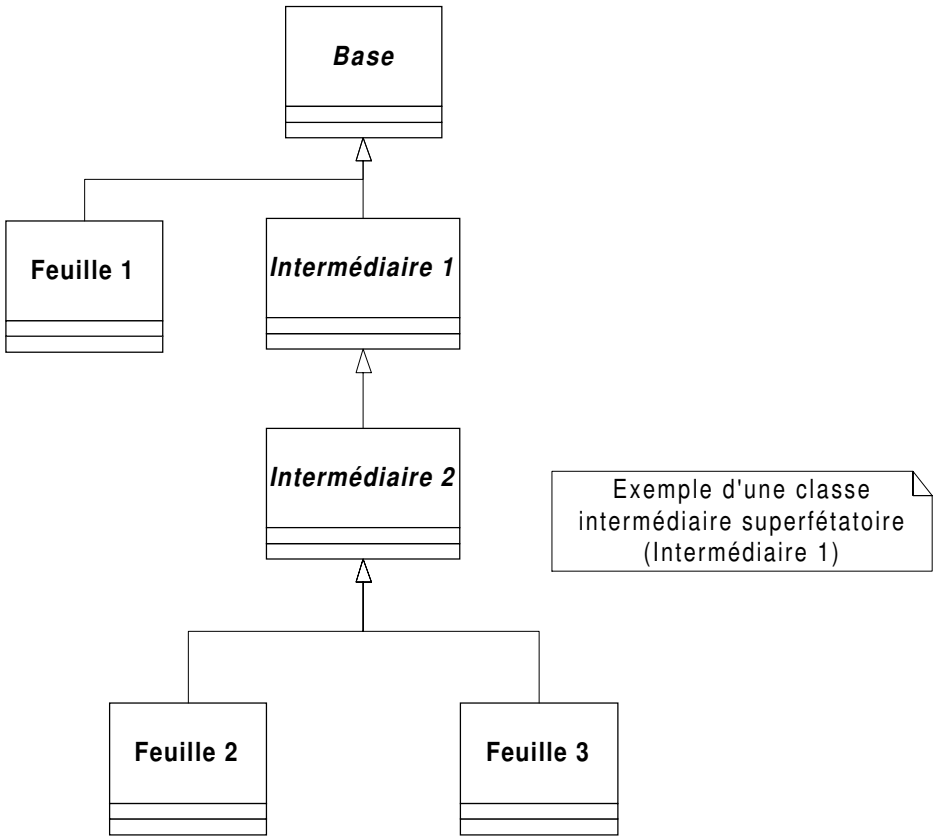

Figure 3.4 Hiérarchie contenant une classe inutile

3.4.2 L'héritage de construction

L'héritage de construction est un autre exemple de mauvaise utilisation de l'héritage à bannir absolument. En effet il consiste à dériver une classe en lui ajoutant de nouveaux attributs de manière à modifier totalement le concept utilisé. Par exemple, cela revient à considérer qu'un rectangle est une ligne auquel on a rajouté une deuxième dimension. En outre, dans notre dernier cas, il est facile de voir que l'héritage est impropre, car la phrase "Un rectangle est une version spécialisée d'une ligne" n'a pas de sens.

3.4.3 Les incohérences conceptuelles

Parfois l'héritage peut conduire à des aberrations conceptuelles. Considérons par exemple une classe oiseau pourvue de la méthode Voler. On peut, par exemple dériver la classe Pingouin qui sera elle même pourvue de la méthode Voler, soit par héritage, soit par redéfinition alors que chacun sait très bien que les pingouins ne volent pas même si, dans ce cas, on peut dire sans crainte "Un pingouin est un oiseau spécialisé". Aussi, certains langages proposent de l'héritage sélectif : le programmeur est invité à spécifier quels attributs et quelles méthodes seront héritées. Hors supprimer la méthode Voler dans la hiérarchie de Oiseau n'est pas sans poser de problème. En effet, cela nuit totalement au principe de polymorphisme qui veut qu'une méthode présente dans la classe de base puisse toujours se retrouver dans les classes dérivées.

3.5 L'héritage multiple

L'héritage multiple est une extension au modèle d'héritage simple où l'on autorise une classe à posséder plusieurs classes mères afin de modéliser une généralisation multiple.

Par exemple, supposons que l'on souhaite ajouter la classe `Hovercraft` à notre modèle de parc de véhicules. Un Hovercraft est à la fois un bateau et un véhicule terrestre. Aussi, il est possible de le modéliser de la manière suivante :

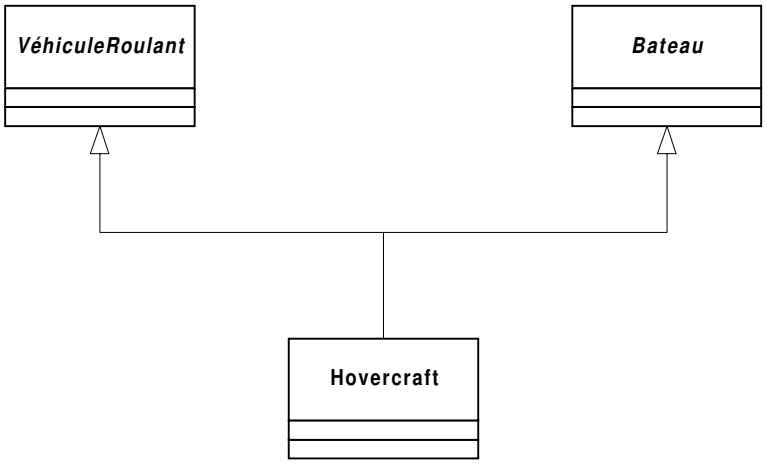

Figure 3.5 Exemple d'utilisation de l'héritage multiple

L'utilisation de l'héritage multiple n'est pas sans poser certains problèmes. Par exemple, si les deux classes de base ont des attributs ou des méthodes de même nom, on se trouve confrontés à des collisions de nommage qu'il convient de résoudre. Certains langages préfixent le nom de l'attribut par sa classe d'origine.

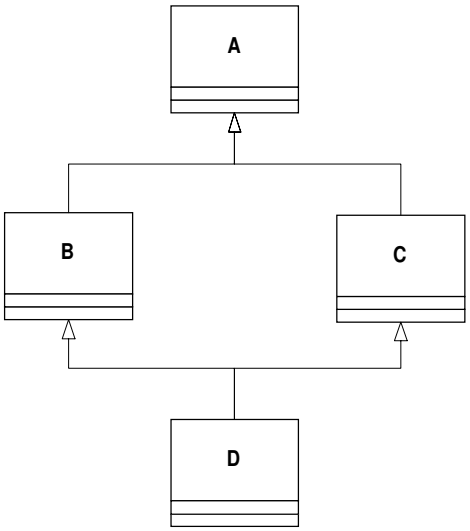

Figure 3.6 Héritage à répétition : D reçoit deux copies de sa classe ancêtre A

Autre problème (illustré par la Figure 3.6) : l'héritage multiple se transforme très souvent en héritage à répétition. Ici la classe D hérite de deux copies de la classe A, une à travers la classe B, l'autre à travers la classe C. Quelle copie doit être conservée ? Certains langages (dont le C++) proposent des mécanismes permettant de

gérer cette situation épineuses susceptible d'arriver, par exemple, si toutes les classes dérivent d'une même classe ancêtre.

De nombreuses bibliothèques de classes utilisent une classe abstraite de base afin de bénéficier de mécanismes polymorphiques. Aussi, l'on risque d'être confronté au problème de l'héritage à répétition dès que l'on utilise de l'héritage multiple. Les interfaces fournies par des langages bannissant l'héritage multiple (tels que Objective C ou JAVA) proposent une alternative intéressante et particulièrement agréable à l'héritage multiple.

3.6 Les interfaces

Les problèmes liés à l'héritage multiple ont conduit certains concepteurs de langage à le bannir des fonctionnalités proposées. On a même vu l'apparition d'autres mécanismes, tels que les *interfaces*. Une interface est semblable à une classe sans attribut (mais pouvant contenir des constantes) dont toutes les méthodes sont abstraites.

En plus de ses caractéristiques d'héritage, une classe implémente une interface si elle propose une implémentation pour chacune des méthodes décrites en interface. Ce mécanisme est particulièrement puissant car il crée des relations fortes entre différentes classes implémentant les mêmes interfaces sans que ces dernières aient une relation de parenté. En particulier, les méthodes décrites dans les interfaces sont, par définition, polymorphes puisqu'elles sont implémentées de façon indépendante dans chaque classe implémentant une même interface.

En outre, chaque classe peut implémenter autant d'interfaces qu'elle le désire. Ce mécanisme, issu du Smalltalk a été repris par les langages Objective C et Java en particulier.

Afin de fixer les idées, considérons, par exemple, un système d'objets modélisant un sous marin nucléaire dont certaines composantes appartenant à des branches hiérarchiques doivent pouvoir être affichées sur un plan. Plutôt que de faire dériver toutes les classes d'un ancêtre proposant des méthodes d'affichage ce qui serait conceptuellement mauvais – certaines classes n'ayant pas lieu d'être affichées, il sera préférable d'adjoindre une interface `Affichable` à celles devant figurer sur le plan. Notons également que si toutes les classes dérivaient d'une classe de base proposant les méthodes de dessin, toutes les classes non graphiques auraient été alourdies de diverses méthodes inutiles.

Faire la différence entre héritage et utilisation d'interfaces n'est pas toujours aisé et dépend du point de vue du concepteur mais également de l'environnement du logiciel à construire. En effet, considérons à nouveau l'exemple de l'hovercraft. Faut il le faire dériver uniquement de `MachineRoulante` et lui faire implémenter une interface `MachineNavigante` … ou bien réaliser l'héritage sur `Bateau` et faire implémenter une interface `MachineTerrestre` ou encore, créer une nouvelle classe indépendante et lui faire implémenter les deux interfaces. La réponse à la question n'est pas si simple. En effet, cela dépend des priorités que l'on souhaite donner dans le modèle : le critère le plus prioritaire devant être le siège de l'héritage. Une fois de plus se pose le problème de la subjectivité de la décomposition qui dépend fortement de la personnalité du modélisateur et de l'environnement du logiciel à concevoir.

4. L'agrégation

4.1 Définition

L'agrégation est un autre type de relation entre deux classes qui traduit cette fois les relations *« Est composé de ... »* ou *« Possède … »* ou encore *« a … »*. Par exemple, dans un système mécanique, on pourrait considérer que la classe `Voiture` est composée d'une instance de la classe `Moteur`, quatre instances de la classe `Roue` et une instance de la classe `Chassis`. L'instanciation passe nécessairement pas l'utilisation d'attributs qui sont eux mêmes des objets ou des pointeurs sur des objets ou même une instance d'une classe conteneur qui elle réalise effectivement l'agrégation. L'une des caractéristiques principale de l'agrégation est sa *cardinalité*. Considérons par exemple l'agrégation de roues par une voiture. Une voiture possède exactement 4 roues (je laisse la roue de secours et les voitures moisies à 3 roues au rencard) et chaque roue ne peut pas être possédée par plus d'une voiture. La cardinalité de l'agrégation est donc de 1 côté agrégateur et de 4 du côté agrégé.

La figure suivante permettra de fixer les idées :

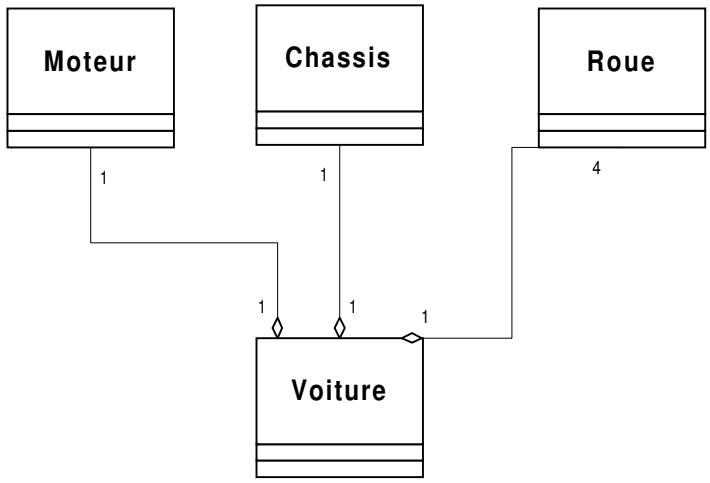

Figure 4.1 Exemple d'agrégation pour une classe Véhicule

La notation UML utilise une flèche dont la pointe est un losange pour représenter l'agrégation. Le losange est du côté de l'agrégateur (de la classe composée). Les cardinalités sont indiquées :

- à côté du losange pour la cardinalité de l'agrégateur (sur le schéma : une roue appartient à une seule voiture)
- à côté du trait pour la cardinalité de l'agrégé (sur le schéma : une voiture possède exactement 4 roues)

4.2 L'agrégation comme alternative à l'héritage multiple ou aux interfaces

Il est parfois possible de traduire en termes d'agrégation des notions apparentées à l'héritage multiple ou à l'utilisation d'interfaces. Considérons un système militaire regroupant des avions et des radars. Tout d'un coup, on décide d'adjoindre un AWACS : système mixte Avion / Radar. Comment modéliser la situation ?

1. Héritage multiple : dériver la classe AWACS d'Avion et de Radar.

2. Utilisation d'interfaces :

   2.1. Dériver `AWACS d'Avion` et lui adjoindre une interface `Détecteur` implémentée par tous les objets de type `Radar`

   2.2. Dériver `AWACS` de `Radar` et lui adjoindre une interface `MachineVolante` implémentée par tous les avions du modèle

   2.3. Créer une classe `AWACS` indépendante et lui adjoindre les interfaces `Détecteur` et `MachineVolante`

3. Utilisation d'agrégation :

   3.1. Dériver `AWACS d'Avion` et lui ajouter un attribut `Radar`

   3.2. Dériver `AWACS` de `Radar` et lui ajouter un attribut `Avion`

   3.3. Créer une classe `AWACS` indépendante et lui ajouter deux attributs : un `Radar` et un `Avion`

4. ...

Ces différentes possibilités montrent à la fois l'étendue exceptionnelle de la richesse expressionnelle de l'objet et ces faiblesses. En effet, si certains modèles paraissent intuitivement meilleurs que d'autres, aucun ne ressort franchement du lot comme étant le meilleur. En outre, il était également possible d'utiliser en même temps agrégation et implémentation d'interface là où nous avons choisi héritage et agrégation.

En outre, un modélisateur orienté Radar préférera probablement les modèles 1, 2.2 et 3.2 alors qu'un constructeur d'avions mettra l'accent sur son produit de prédilection en privilégiant les dérivations d'avion 1, 2.1 et 3.1.

Les schémas de la page suivante montrent certains des différents modèles que l'on peut créer.

> A l'heure actuelle, il n'existe pas de notation clairement définie en UML pour représenter les interfaces. Aussi, le moyen le plus simple consiste à utiliser une classe déclarée abstraite (sans attributs !) et dont le nom est préfixé du mot Interface.

Nous utiliserons une flèche semblable à celle représentant la généralisation / spécialisation mais accompagnée d'un trait pointillé pour symboliser l'implémentation d'une interface par une classe ; la pointe de la flèche étant dirigée vers l'interface.

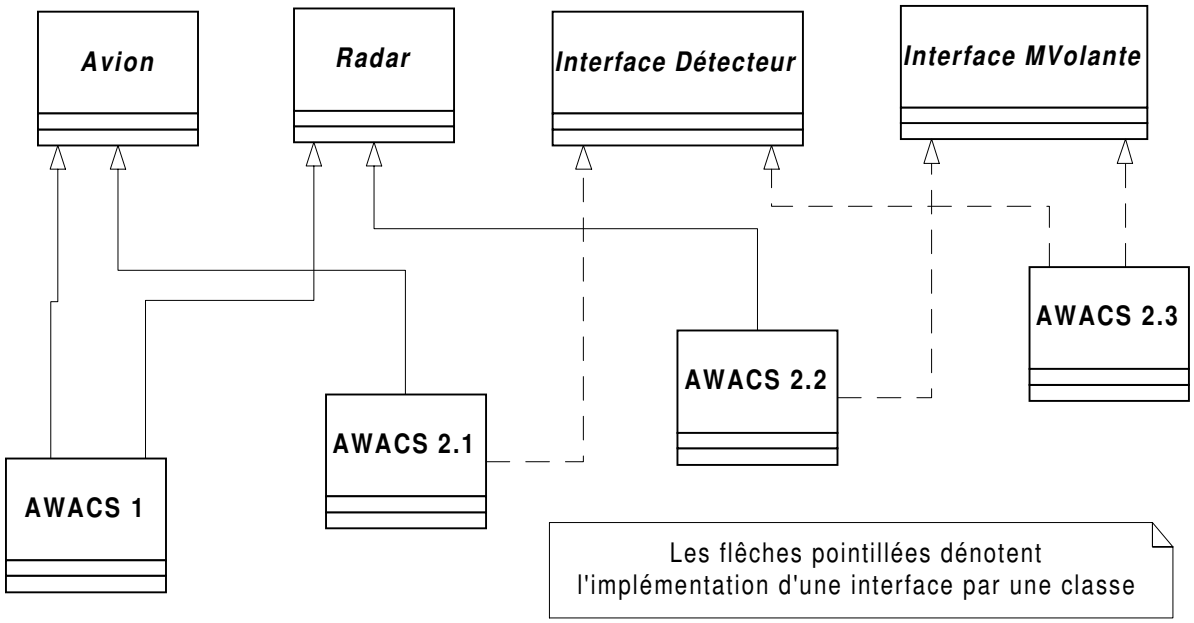

Figure 4.2 Modélisations de l'AWACS mettant en oeuvre des interfaces

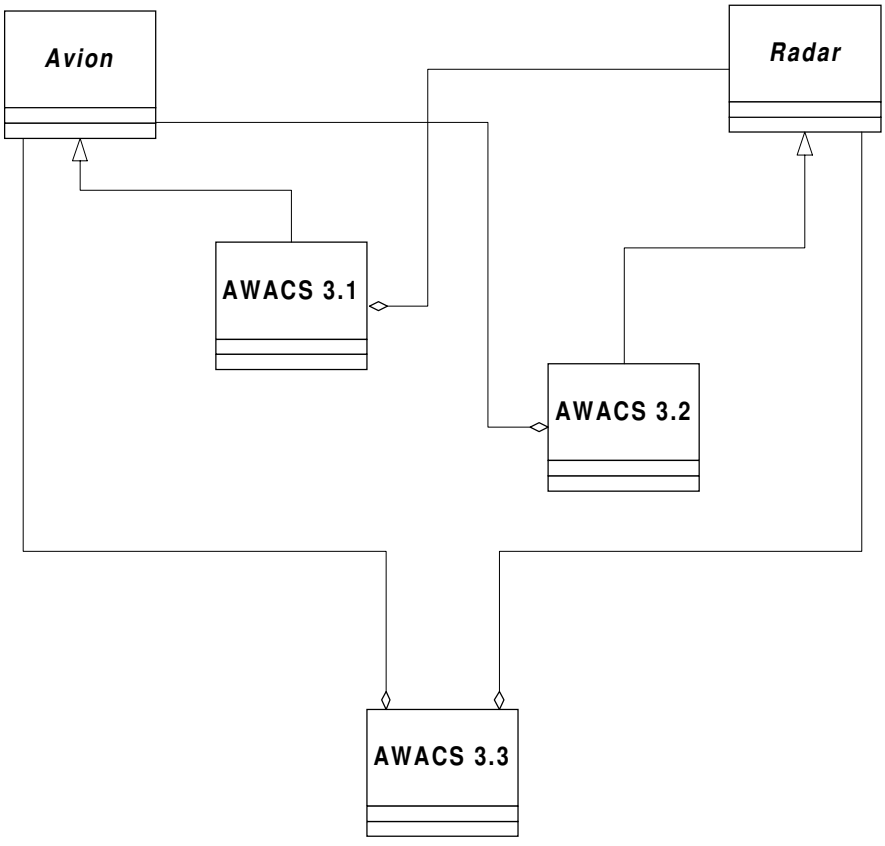

Figure 4.3 Modélisations de l'AWACS utilisant de l'agrégation et de l'héritage

5. Le polymorphisme

5.1 Définition

Le polymorphisme est le troisième des trois grands principes sur lequel repose le paradigme objet. C'est assurément son aspect à la fois le plus puissant et le plus troublant. Comme son nom l'indique le polymorphisme permet à une méthode d'adopter plusieurs formes sur des classes différentes. Selon les langages, le polymorphisme pourra s'exprimer sur l'ensemble des classes d'un système alors que d'autres le confinent aux classes appartenant à une même hiérarchie.

5.2 La puissance du polymorphisme

Nous allons démontrer la puissance du polymorphisme au travers de l'exemple classique des classes d'objets graphiques. Un document dessin peut être vu comme une collection d'objets graphiques qui va contenir des cercles, des rectangles, des lignes, ou toute autre sorte d'objet graphique qui pourrait dériver de la classe `ObjetGraphique`. Une telle agrégation est rendue possible par la notion de *compatibilité descendante des pointeurs*. En effet, un pointeur (ou, dans certains langages, une référence) sur un objet d'une classe spécialisée peut toujours être affecté à un pointeur sur un objet d'une classe généraliste.

Si nous voulons dessiner un dessin tout entier, il nous faudra appeler la méthode Afficher pour chacun des objets contenus dans le dessin. Hors, nous avons pris soin de conserver la même signature pour les différentes méthodes Afficher de tous les objets appartenant à la hiérarchie `d'ObjetGraphique` : c'est la condition Sine Qua Non de l'utilisation du polymorphisme. En effet, nous pouvons maintenant utiliser un code du style :

```
Méthode Dessin::Afficher
{
  Pour chaque Objet inclus
  {
    [objet Afficher]
  }
}
```

Programme 5.1 Utilisation du polymorphisme sur une collection

Le polymorphisme de la méthode `Afficher` garantit que la bonne méthode sera appelée sur chaque objet. La mécanique interne de ce mécanisme stupéfiant repose sur la stratégie de liaison différée ou *Late Binding*. Considérons un programme classique, l'adresse d'appel d'une procédure ou d'une fonction est calculée au moment de l'édition de liens et codée en dur dans le programme : c'est la liaison initiale (ou *Early Binding*). Dans le cas de la liaison différée, l'emplacement de la méthode à appeler est situé dans l'objet lui même. C'est donc à l'exécution que le programme va établir l'adresse d'appel.

Un autre exemple est celui de la méthode `DéplacerVers` (voir le Programme 3.1) que nous avons explicité ci-dessus. En effet, ce code est valable quelle que soit la classe de l'objet graphique sur lequel on l'applique : un déplacement consiste toujours en un effacement préalable, une modification des coordonnées puis un affichage. Une

fois encore, ce code fonctionne grâce au polymorphisme car il est nécessaire que les bonnes versions de `Effacer` et `Afficher` soient appelées.

De même, si l'on considère que l'effacement consiste à réafficher un objet dans la couleur du fond, on pourrait définir la méthode `Effacer` comme suit :

```
Méthode ObjetGraphique::Effacer
{
  [objet setCouleur : couleurFonds]
  [objet Afficher]
}
```

Programme 5.2 Utilisation du polymorphisme dans la méthode `Effacer`

5.3 Une forme faible de polymorphisme : la surcharge

La surcharge est un mécanisme fréquemment proposé par les langages de programmation orientés objet et qui permet d'associer au même nom de méthode / fonction / procédure différentes signatures.

Par exemple, on pourrait proposer deux signatures différentes pour la méthode `Afficher` :

Pas d'argument si l'on désire utiliser le périphérique d'affichage par défaut

Spécification d'un périphérique en argument

Ce mécanisme n'est qu'une aide à la

6. La relation d'association

L'association est la troisième grande forme de relation que nous allons considérer après celles d'héritage et d'agrégation. Si l'héritage ne souffre d'aucune ambiguïté car elle traduit la phrase « est une forme spécialisée de (IS A) ». La relation d'association est elle plus difficile à caractériser. En effet, selon les auteurs, elle peut être « Communique avec » ou bien « Utilise un » (USES A). En fait, et dans certains cas, il est même facile de la confondre avec la relation d'agrégation comme nous allons le voir dans l'exemple qui suit.

Afin de fixe un peu les idées, considérons à nouveau l'exemple classique du Zoo. D'un certain point de vue (rappelons au passage que le principe d'abstraction est subjectif et dépend fondamentalement du point de vue du modélisateur), le Zoo « est composé de »:

- Un ensemble de cages
- Un ensemble d'animaux
- Un ensemble de gardiens

Ces relations étant, de toute évidence, de l'agrégation !

En revanche, un gardien doit s'occuper d'un certain nombre d'animaux (possédés, par le Zoo) et nettoyer un certain nombre de cages (possédées par le Zoo). De la même manière, une cage regroupe certains animaux (possédés par le Zoo).

Ces dernières relations ne sont pas de l'agrégation (on considère habituellement qu'un même objet n'est pas agréable par plusieurs) mais plutôt des Associations. En fait, on rangera dans l'association tout type de relation un peu flou qui ne sera ni de l'agrégation, ni de l'héritage. On obtient alors le schéma de la Figure 6.1.

A l'instar de l'agrégation, on définit des cardinalités sur la relation d'association ainsi que des rôles. Par exemple, si nous considérons la relation entre les classes Cage et Gardien, nous pouvons lire :

"Un Gardien nettoie de 0 à n Cages / Une cage est nettoyée par 1 et un seul gardien"

Toutefois, il est souvent difficile de définir ce qui est de l'agrégation ou de l'association. Par exemple, un autre modèle consisterait à considérer que le zoo agrège les gardiens et les cages et que ces dernières agrègent les animaux. Ce cas de figure est présenté sur la Figure 6.2.

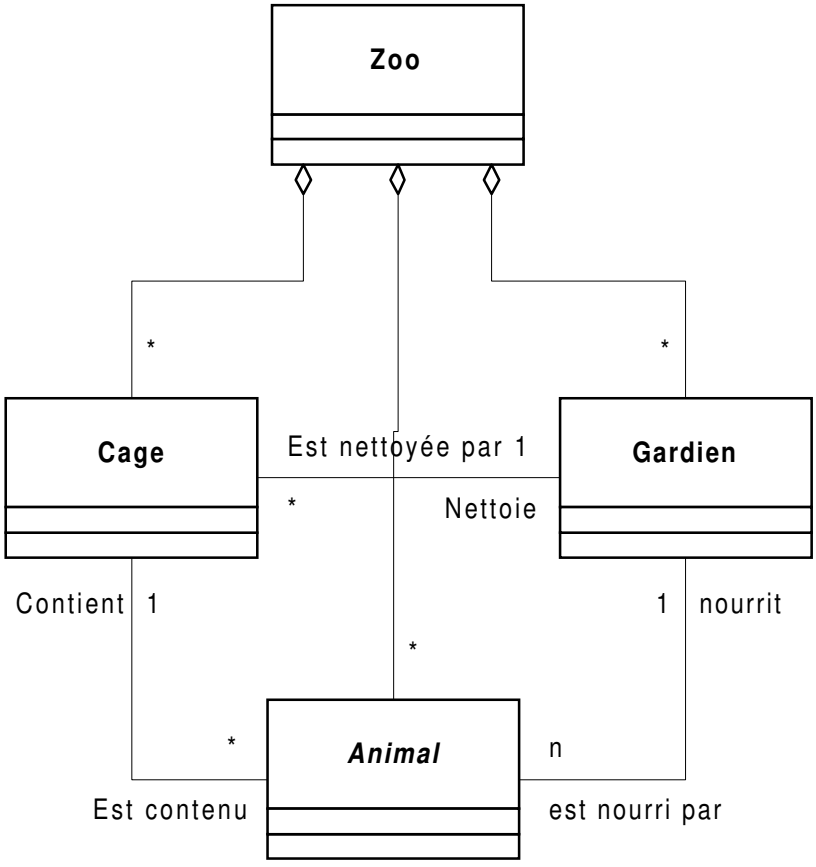

Figure 6.1 Modélisation du Zoo par agrégation et association

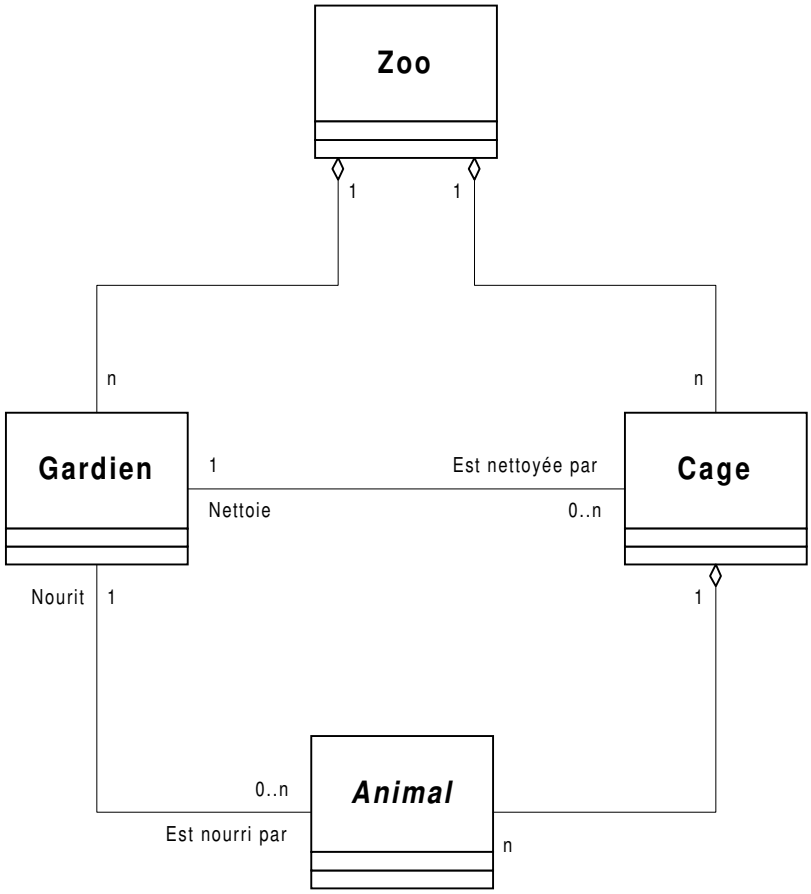

Figure 6.2 Autre modélisation du Zoo par agrégation et association

7. Pour conclure sur le modèle objet

Nous n'avons montré ici que quelques uns des concepts du modèle objet qui est en réalité beaucoup plus vaste. Les relations d'agrégation, association et héritage sont les plus fondamentales. Certains auteurs considèrent que l'on peut tout faire à partir de ces trois motifs essentiels, certains préconisent d'autres types de relations. Le point essentiel à retenir est que la modélisation repose sur le principe d'abstraction qui est fondamentalement subjectif et fortement lié au modélisateur et aux objectifs du logiciel que l'on cherche à concevoir ! Ceci n'est pas limité aux objets, mais à tout processus de modélisation mais peut ici peut prendre une ampleur considérable.

Afin de terminer sur une note positive, examinons la structure d'un programme construit selon les principes objet : un tel logiciel est une collection d'objets communiquant par messages et en interaction constante. Ceci permet d'introduire un nouveau principe d'abstraction : l'abstraction d'exécution. En effet, les objets permettent de masquer l'organisation d'un logiciel. Les différents constituants peuvent s'exécuter de façon séquentielle ou concurrente (avec diverses méthodes de communication et ou synchronisation), être situés sur une même machine ou distribués sur un réseau : le logiciel final est toujours vu du point de vue de l'utilisateur comme un objet monolithique !

8. Glossaire

Abstraction de données	Principe considérant que la structure interne des données est cachée à leur utilisateur qui ne connaît d'elles que les procédures permettant de les traiter. Du point de vue de la programmation objet, on considérera toujours un objet comme une boîte noire que l'on manipule en lui envoyant des messages.
Abstraction procédurale	Principe considérant que toute action est atomique du point de vue de celui qui la déclenche. En un mot, il n'a aucune idée des mécanismes mis en jeu dans l'implémentation de l'action. Du point de vue de la programmation orientée objet, cela signifie que l'appel d'un message est toujours vu comme atomique. Dans la pratique cela revient à masquer à l'utilisateur le code d'implémentation des sous programmes qu'il utilise.
Classe	Composant logiciel décrivant des objets. On dit que la classe est la méta-donnée des objets
Classe dérivée	Voir sous classe
Classe fille	Voir sous classe
Constructeur	Méthode spéciale, à mi chemin entre la méthode de classe et la méthode d'instance chargée de créer un nouvel objet en mémoire et d'initialiser son état.
Encapsulation	L'un des trois principes fondamentaux du paradigme objet. Prône, d'une part le rassemblement des données et du code les utilisant dans une entité unique nommée objet, d'autre part, la séparation nette entre la partie publique d'un objet (ou interface) seule connue de l'utilisateur de la partie privée ou implémentation qui doit rester masquée. Pour de plus amples explications
Héritage	L'un des trois principes fondamentaux du paradigme objet. Possibilité offerte par les langages orientés de définir des arborescences de classes traduisant le principe de généralisation spécialisation. Une relation d'héritage peut se traduire par la phrase : *« La classe dérivée est une version spécialisée de sa classe de base »*
Logiciel orienté objet	Nouvelle façon de concevoir le logiciel orientée vers les données plutôt que sur les actions, les données ayant une plus longue pérennité conceptuelle. Un logiciel est alors vu comme une collection d'objets communiquant par messages.
Message	Unique moyen de communication fourni par les objets. Une invocation de message se traduit habituellement par l'activation d'une méthode. D'ailleurs, dans la plupart des langages orientés objet modernes, il n'y a pas de distinction entre les notions de message et de méthode
Objet	Entité fondamentale rassemblant des données ainsi que le code opérant sur ces données. Un objet communique avec son environnement par messages et répond aux principes d'abstraction de données et d'abstraction procédurale. Un ensemble d'objets présentant les mêmes caractéristiques forme une classe.

Paradigme objet	Nouvelle façon de concevoir le logiciel. L'accent est mis sur les données et les actions qu'elles supportent plutôt que sur une vision totalement procédurale. Le paradigme objet est basé sur trois principes fondamentaux : l'encapsulation, l'héritage et le polymorphisme.
Polymorphisme	L'un des trois principes fondamentaux du paradigme objet. Faculté présentée par certaines méthodes dont le comportement est différent (malgré une signature identique) selon l'objet auquel elle s'appliquent. Dans la plupart des langages orientés objet, ce comportement n'est disponible que pour des classes appartenant au même graphe d'héritage.
Signature	Liste de paramètres et type de retour d'une méthode / procédure / fonction.
Sous classe	On dit également Classe fille, ou classe Dérivée. Se dit d'une classe qui dérive d'une autre classe (on parle de classe mère ou de super classe). C'est une forme spécialisée de sa super classe et à ce titre, elle redéfinit souvent certaines ce ces méthodes pour les adapter à ces fonctions. Voir Héritage.
Super classe	Classe générale qui a été dérivée en une ou plusieurs sous classes. Une super classe définit un cadre de travail et propose des méthodes d'ordre général à toute une famille. Certaines d'entre elles sont destinées à être redéfinies par les sous classes afin d'adapter leur comportement dans une classe spécialisée. Une super classe peut même être *abstraite*, c'est à dire ne pas fournir d'implémentation pour certaines méthodes qui devront alors être redéfinies. Voir Héritage.