
1

Concepts et Formalismes UML

www.thierrycros.net

1

Concepts et notations UML

1.1 Le concept d'objet

Le formalisme UML repose sur le concept d'objet. Or, cette approche est apparue dans les langages avant d'être formalisée dans UML. En ce qui concerne le langage de programmation, un objet est le regroupement d'attributs et de méthodes. On peut alors considérer l'objet comme un ensemble de fonctions (les fonctions membres de C++) qui partagent quelques variables. Un objet, depuis une perspective donnée, est un ensemble de champs munis d'opérations applicables sur ces informations. Toutes ces définitions souffrent d'un attachement à la programmation. Un langage comme le C++, qui permet d'obtenir de bons programmes objet, ne facilite pas la prise d'altitude : il autorise la création de `struct` qui comportent des champs fonction.

Un objet est un élément d'un système, identifié, distinct et insécable, doué de comportements qui dépendent de son état.

Cette définition, première clé des concepts, est indépendante de toute considération informatique, d'où l'application possible au « non-informatique ». Un système est un ensemble d'éléments (ou objets) en relation. Cette caractérisation correspond à la vision objet que l'on doit acquérir en développement de logiciels.

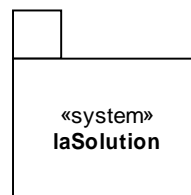


Figure 1-1 : Représentation UML du paquetage que constitue le système dans son ensemble.

Cette nouvelle perspective décuple le potentiel de ce concept : d'une part UML (grâce à l'objet) modélise de nombreux types de systèmes indépendants du développement

informatique ; d'autre part l'universalité de ces concepts permet de les utiliser depuis la phase d'expression de besoins jusqu'à la phase d'implémentation. Ainsi, UML permet de modéliser des organisations, des métiers (business modeling), des domaines (domain modeling), l'expression de besoins, l'implémentation sous forme de composants tels que fichiers « source » et exécutables.

Un objet est distinct dans le système. Cette propriété induit son identité. En activité d'analyse en particulier un objet n'a pas à recevoir d'identifiant. D'un point de vue pratique, cet identifiant intrinsèque correspondra à son adresse en mémoire vive ou bien à un numéro implicite d'enregistrement dans un fichier. Bien souvent, ces identifiants doivent être remplacés par des clés primaires en activité de conception.

Les objets sont insécables. Supprimer une opération ou une partie de l'état dénature l'objet. Cette propriété est à retenir en conception objet : Grady Booch précise¹ qu'un objet doit être *complet*.

La définition met l'accent sur la dynamique. Le comportement est l'effet observable de la réception d'un message par un objet. Un message est un stimulus envoyé à l'objet. Autrement dit, un objet est par essence dynamique. Certains les considèrent comme de petits êtres vivants. Sans exagérer outre mesure cette analogie, elle a le mérite de préciser la nature d'un objet dans un système. Un objet ne subit pas des fonctions comme une structure de données. Au contraire, il reçoit des stimuli, et, en fonction de l'état dans lequel il se trouve, il décide de déclencher telle ou telle opération. Prenons l'exemple d'un avion dans un système tel qu'un aéroport. Un objet *tour de contrôle* envoie un message, un stimulus, à un objet destinataire avion. Le comportement observable est l'élévation de l'avion dans l'air. Cela est dû au déclenchement d'opérations telles que *décoller()*. Le point remarquable est que l'avion est autonome mais pas indépendant dans le système. Il décolle par ses seuls moyens. Toutefois, il s'inscrit dans le fonctionnement de l'aéroport et ne décolle pas n'importe quand. Au contraire, l'interaction entre la tour de contrôle et l'avion ordonnance le décollage. C'est ce que l'on nomme plus précisément une collaboration, terme fondamental dans l'approche objet.

Enfin l'état de l'objet. L'analogie avec les êtres vivants nous permet de considérer l'état de l'objet, la situation dans laquelle il se trouve, comme sa *mémoire* ou plus précisément le résultat de ses actions passées. L'état d'un objet n'a d'intérêt que dans la perspective d'une future utilisation par une opération. Un objet *facture* mémorise sa date car elle lui sera utile plus tard dans une collaboration d'impression par exemple. La date participe à l'état : lorsque l'événement de demande d'informations nécessaires à l'impression parvient à l'objet, son état lui permet de le traiter correctement, par exemple au travers d'un accesseur qui fournit une copie de cet attribut. Sans l'état, un objet serait un groupement de traitements. L'état conditionne les opérations. *A contrario*, un état constamment inutilisé par les opérations ne sert à rien. Ainsi comportement et état sont intimement liés. Malheureusement, des dizaines d'années de dichotomie données / fonction ne facilitent pas cette vision réunifiée de l'objet. L'état est donc plus que les valeurs d'attributs de l'objet. Il représente aussi un potentiel de comportements dans la mesure où il autorise ou non le déclenchement d'opérations. Lorsque l'avion est dans un état « en vol », le comportement qui consiste à embarquer des passagers n'est pas autorisé.

¹ Analyse et conception orientées objets chapitre 3. « Un objet... possède toutes les caractéristiques pertinentes de l'abstraction. »

UML représente un objet par un rectangle dont le label est souligné.

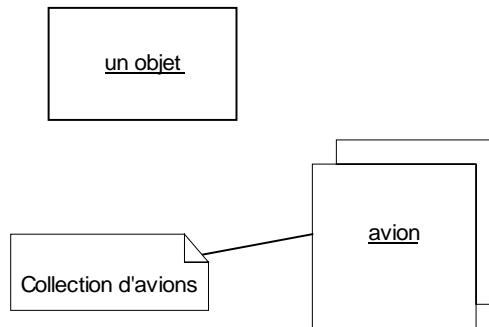


Figure 1-2 : Un objet en UML

Le concept d'objet est le concept essentiel avec celui d'interaction. Le système est en résumé un volume, un espace composé d'objets prêts à réagir au moindre stimulus. L'activité d'analyse permet de structurer les objets du système. L'activité de conception définit les objets à implémenter.

1.2 Les interactions entre objets

Les interactions entre objets forment la deuxième clé d'une véritable approche objet. Les objets interagissent ou collaborent. Ainsi, le fonctionnement du système est obtenu parce que différents éléments, les objets, participent à l'effort commun. UML représente une interaction en terme d'objets collaborants et de messages envoyés. Un objet émetteur envoie un message à un objet destinataire. Pour cela, les deux objets doivent être « connectés ». Autrement dit, un lien, en tant que support logique d'envoi de message, doit les relier. « Support logique » signifie que le lien n'est rien d'autre que la relation logique qui permet le dialogue entre les objets. Les liens sont de multiples natures (voir par exemple les stéréotypes de liens dans les diagrammes de collaboration).

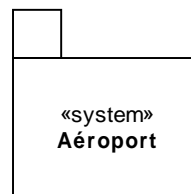


Figure 1-3 : Le système aéroport

Le système « Aéroport » fonctionne car un objet *tour de contrôle* envoie des messages à des objets avions. Dans ce cas, le lien entre les objets indique la connaissance qu'a l'objet *tour de contrôle* des objets avions. Supposons un scénario dans lequel la *tour de contrôle* gère deux avions. Elle provoque le décollage du premier puis du second.

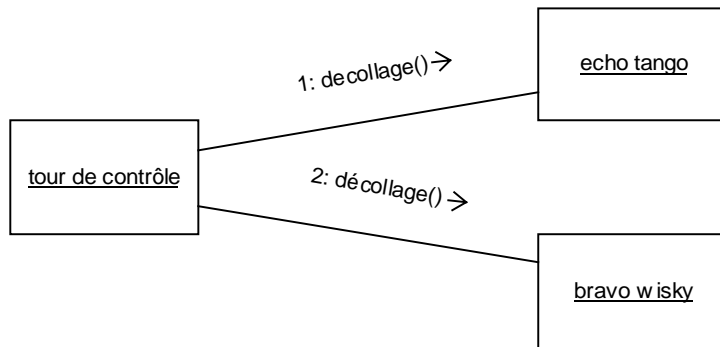


Figure 1-4 : Collaboration entre objets d'un aéroport.

Dans ce diagramme de collaboration, trois objets sont représentés par trois rectangles. Ce sont les premières notations à décoder. Deux liens indiquent un potentiel de communication : tour de contrôle et premier avion, tour de contrôle et deuxième avion. Le diagramme n'indique pas de lien entre les deux avions : ils ne peuvent donc pas communiquer directement. A ce niveau, le schéma est un diagramme d'objets.

Les messages (décollage) sont fléchés pour indiquer leurs sens. Ils sont aussi numérotés, ce qui précise le séquençement d'envoi. L'objet `tour de contrôle` joue un rôle particulier. Son nom indique qu'il contrôle, ordonnance, supervise d'autres objets : les avions. Cette situation est tout à fait classique en objet : des objets contrôleurs sont injectés dans les systèmes pour prendre en charge le déroulement des interactions. C'est leur responsabilité par rapport aux autres éléments du logiciel. Contrairement à une approche fonctionnelle, le contrôle réside dans certains objets, pas dans les sous-programmes.

Un message est implémenté sous forme d'appel de fonction membre, d'interruption logicielle ou matérielle, d'événement graphique, etc. La syntaxe d'un langage tel que le C++ ne doit pas sous-informer quant à la réalité du système. Reprenons l'exemple de l'aéroport. Une syntaxe possible, en C++, d'implémentation de la collaboration de la figure 1-3 est la suivante.

```

Avion premierAvion ;
Avion deuxièmeAvion ;
TourDeContrôle tour ;
...
premierAvion.décollage() ;
deuxièmeAvion.décollage() ;
  
```

Une vision objet de notre programme traduit l'instruction `premierAvion.décollage()` par : envoi du message `décollage` à l'objet `premierAvion`.

1.3 Les classes

1.3.1 Classes et interfaces

Les systèmes sont constitués de nombreux objets identiques. Ils sont regroupés en classes. Autrement dit, ce concept est une aide à la modélisation du système : les classes sont nécessaires pour modéliser et donc mieux comprendre un système, mais elles ne représentent pas *l'essence* du système qui réside dans les objets. Ces derniers sont la finalité de l'obtention des classes. UML représente une classe par un rectangle muni éventuellement de plusieurs compartiments. Les notations classiques en utilisent généralement trois : nom, attributs, opérations. Il est possible de compléter ce formalisme de base par des compartiments **exceptions** et **responsabilités**. Si l'atelier de génie logiciel (AGL) n'offre pas ces représentations, il est possible de les simuler par des notes stéréotypées ou de les restituer sous forme de propriété de la classe.



Figure 1-5 : Représentation minimale d'une classe en UML : uniquement le nom.

Le degré de connaissance du système, le point de vue que l'on souhaite restituer sont autant de bonnes raisons de visualiser uniquement le compartiment nom de la classe. Une étude partielle de la classe avion permet d'obtenir la représentation suivante.

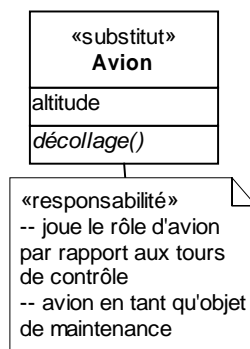


Figure 1-6 : Représentation d'une classe décorée par un stéréotype et un commentaire.

Un stéréotype <<substitut>> (personnalisation du langage par stéréotypage) indique que cette classe est une abstraction ou représentation d'objets réels du domaine.

N Les substituts du monde réel, expression synonyme de type abstrait en pratique, existaient dans la méthode
 O OMT (Object Modeling Technique). Cette méthode proposait plusieurs stéréotypes, sans préciser à l'époque
 T qu'il s'agissait de stéréotypes. Les objets du domaine (les substituts) qui correspondent aux objets métier, les
 E objets de l'application tels que présentation (miroir des objets métier), contrôleur (séquencement des interactions), etc.²

² OMT Modélisation et conception orientées objet, James Rumbaugh et al. Page 472.

Au contraire, une classe telle que FenêtreSaisie est dite applicative car elle n'a aucune contrepartie dans la réalité du domaine. Le commentaire stéréotypé <<responsabilité>> permet de préciser le rôle de la classe dans le système. Les démarches précisent l'ordre d'obtention de ces informations. Par exemple, Object Modeling technique (OMT) propose d'obtenir au préalable les attributs. L'approche Class Responsibilities Collaborators Cards (CRC Cards) insiste sur les responsabilités. Une approche plus comportementale jouerait sur les messages reçus par les objets.

Un ensemble de déclarations d'opérations logiquement regroupées est une **interface**. UML distingue interface et classe, ce qui est aussi le cas du langage Java (mots clé `interface` et `class`) mais pas du C++.

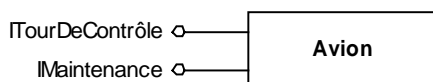


Figure 1-7 : Interface et classe

La ligne qui relie les interfaces (cercles) et la classe (rectangle) correspond à la relation UML de réalisation. Un avion réalise deux vues : ITourDeControle et IMaintenance. Une interface est spécifiée par la liste d'opérations qu'elle représente. Elle doit être réalisée, implémentée par une classe. Les concepts objet supposent une distinction entre interface et implémentation, ce qui correspond en fonctionnel à la distinction déclaration – définition.

```

// Aspect fonctionnel de l'interface
int Fonction(int);           // déclaration : entête suivie de :
int Fonction(int) {...}     // définition : entête suivie de {
  
```

Ainsi, les modules utilisateurs sont découplés des modules réalisateurs. Un module ou unité de compilation client de la fonction est recompilé uniquement si la déclaration évolue, pas lorsque la mise en œuvre elle-même évolue, ce qui limite considérablement les temps de compilation.

1.3.2 Attributs et opérations

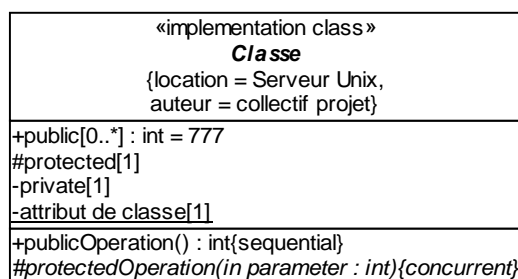


Figure 1-8 : Eléments de modélisation d'une classe

La classe de la figure ci-dessus est abstraite (nom en italique). Une classe peut être détaillée par de nombreux éléments.

Stéréotype : le stéréotype fixe la nature de la classe

Propriétés (valeurs marquées ou nommées)

Attributs, avec leur multiplicité (cas de collections), leurs valeurs initiales, leur accessibilité, leurs portées (instance ou classificateur)

Opérations : paramètres, type de retour, abstraite ou concrète (avec méthode), concurrence (séquentiel, gardé, concurrent).

1.4 Associations

1.4.1 Association



Figure 1-9 : Association entre deux classes.

De même que les objets sont regroupés en classes, de même les liens entre objets sont regroupés en associations. L'association est l'aspect structurel, statique des potentiels d'interactions du système. Si deux objets de deux classes ne communiquent pas, quel que soit le scénario, alors l'association entre ces classes n'a pas de réalité dans le système. Elle peut être sémantiquement correcte, indiquer une relation logique entre les classes, mais cela ne l'empêche pas d'être superflue. Considérer une association en tant qu'ensemble de liens possibles entre les objets permet de mieux appréhender la réalité essentiellement dynamique du système. UML représente une association comme un lien. La distinction réside dans les éléments reliés.

La collaboration entre objets permet l'évolution du système. Les envois de messages étant obtenus grâce aux liens entre objets, les associations entre classes forment la relation majeure du diagramme de classes. Les associations peuvent être nommées par un label. Elles peuvent être décorées par le nom des rôles joués par les classes dans la relation. Le rôle est généralement un substantif, ce qui évite toute ambiguïté avec une tentative de modélisation de la dynamique. L'association est aussi décorée par les multiplicités. La figure 1-3 représente un scénario dans lequel une tour de contrôle est liée à deux avions. Ce scénario définit une cardinalité (nombre d'éléments) possibles de liens entre les objets. Différents scénarios enrichissent les cardinalités des liens entre objets. L'ensemble des cardinalités possibles forme la multiplicité. Cette information est essentiellement fonction de la dynamique au travers des interactions qui mettent en jeu les objets des classes. Le diagramme de classes est alors la compilation de toutes les collaborations qui impliquent la création, l'instanciation, de liens à partir de l'association.

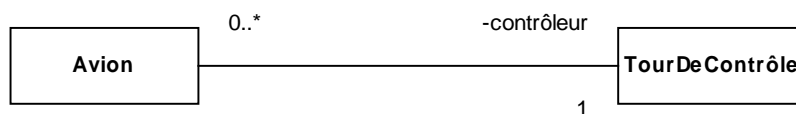


Figure 1-10 : Multiplicité d'association

Les multiplicités suggèrent qu'un avion est toujours contrôlé par une et une seule tour de contrôle. L'objet `tour de contrôle` peut changer dans le temps : un avion est successivement contrôlé par plusieurs tours, ce qui ne peut pas être modélisé dans ce type de diagramme structurel statique.

1.4.2 Agrégation

UML propose une association particulière : l'agrégation, caractéristique de situation de type tout / partie. Elle met en jeu deux classes (minimum) l'une est l'agrégat, le tout, l'autre est le composant, la partie. Cette relation est clairement dissymétrique : le tout est sémantiquement plus « important » que la partie. Un composant peut être partagé par plusieurs agrégats. C'est le cas d'un format de bas de page qui « fait partie » de plusieurs pages à la fois. UML note l'agrégation par un losange blanc du côté agrégat.

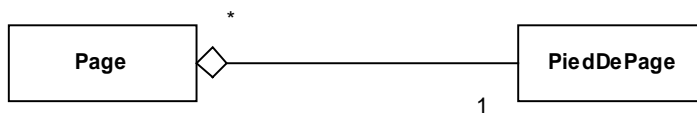


Figure 1-11 : Agrégation : composant partageable.

La figure ci-dessus indique une agrégation entre page et pied de page. La relation d'agrégation est subjective. L'association est au contraire moins ambiguë dans la mesure où elle existe car des liens entre objets existent potentiellement, ce qui ne met pas en jeu la question de l'importance relative des classes. De plus l'implémentation ne distingue pas association et agrégation. La conception pourra donc utiliser l'association sans problème en cas de doute. Elle est probablement moins précise mais a le mérite d'être toujours correcte. Autrement dit, l'agrégation est une affaire de modélisation et constitue une nuance dans les relations entre classes.

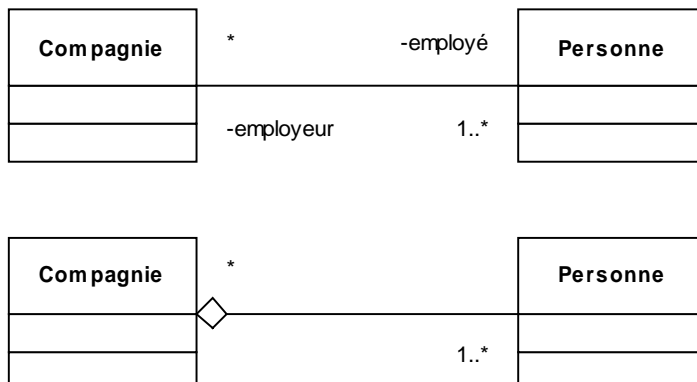


Figure 1-12 : Association ou agrégation ?

La première modélisation suggère la situation (au sens structurel) de collaboration entre les objets des classes : les « collaborateurs » dans l'entreprise. La deuxième modélisation suggère l'appartenance des personnes à l'entreprise.

1.4.3 Composition

Enfin, UML définit une troisième relation associative particulière : la composition. La définition UML de la composition n'est pas celle du langage C++. Une composition est habituellement une agrégation par valeur en C++ : un objet *x* a un attribut *y* qui est instance d'une classe, de même que *n* est instance de *int*. UML propose une définition plus souple qui correspond essentiellement à une agrégation non partageable. Autrement dit, la multiplicité du côté agrégat ou composite ne peut plus être 0..* mais uniquement 0..1. Cette relation est fréquente dans les systèmes. De plus, elle implique une contrainte majeure au niveau du cycle de vie des objets : si un composite disparaît, ses composants aussi.

N
O
T
E

```

Une programmation en C++ de la composition au sens UML pourrait être3 :
class C {
    ~C() { delete composant; };
    Composant composant;
};

```

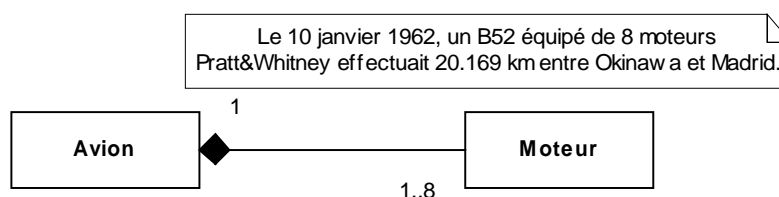


Figure 1-13 : Composition entre classes

1.4.4 Navigation

Une association est un potentiel de liens entre les objets. Un lien est un moyen de communication. De ce point de vue, une association devient un « canal » entre les classes. L'association est navigable : cela traduit le sens d'envoi de messages sur les liens correspondants. Généralement, les agrégations et compositions sont navigables vers le composant.



Figure 1-14 : Navigation

La figure ci-dessus suggère que les pages connaissent leur pied de page et peuvent donc leur envoyer des messages mais pas l'inverse.

³ Merci à Robert C. Martin pour la suggestion.

1.5 Généralisation

1.5.1 Héritage

Cette relation a très certainement contribué, à juste titre, à l'avènement des concepts objet. Elle traduit une situation de type « est-un ». L'héritage s'applique dans les cas de spécialisation, classification ou à l'inverse généralisation. L'héritage induit une réutilisation du développement. Tout ce qui est analysé, conçu, programmé, testé, documenté pour la sur-classe est *ipso facto* disponible pour la sous-classe. Le concepteur de la sous-classe travaille uniquement sur les spécificités de la sous-classe par rapport à la classe de base. UML représente l'héritage par une flèche pointée vers la classe mère.

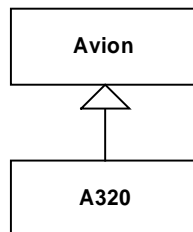


Figure 1-15 : Sur-classe Avion et sous-classe A320.

Lorsque la sur-classe est obtenue avant la sous-classe, l'héritage correspond à une spécialisation ou classification. Ici, un objet A320 est un avion, il s'utilise donc potentiellement en lieu et place d'un avion.

Dans certains cas, quelques classes présentent des membres communs. La généralisation permet leur factorisation et évite ainsi des réécritures inutiles. Certaines méthodes (OMT par exemple) préconisent une recherche de points communs pour réutiliser grâce à l'héritage.

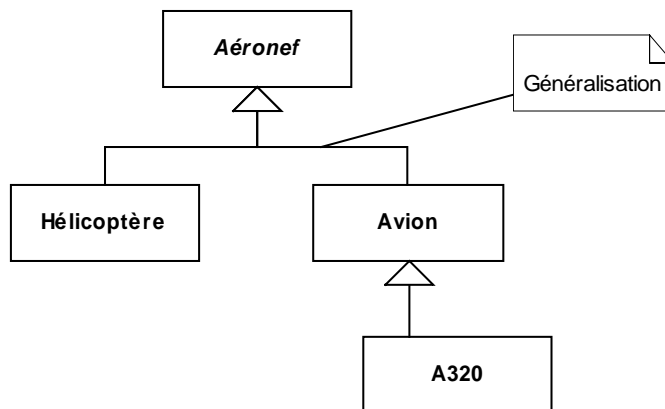


Figure 1-16 : Généralisation

Certains comportements ou attributs communs (`décollage()`, `nombre_passager`) sont reportés au niveau `Aéronef`. Il est à noter que ces classes de haut niveau dans les hiérarchies de classe ne sont pas directement instanciées dans le système. Des avions, des hélicoptères sont dans l'aéroport en tant qu'objets concrets, pas des « aéronefs ». La classe devient abstraite dans le diagramme. D'où son nom en italique. Bien qu'elle n'ait pas de réalité dans le système, la sur-classe abstraite est d'une grande importance dans les taxinomies. En effet, ces classes jouent le rôle de spécification de types. Ici, la classe `Aéronef` n'implémente pas le type, mais elle spécifie ce que doit être un aéronef.

1.5.2 Polymorphisme

Le diagramme de collaboration de la figure 1-3 représente un scénario dans lequel une tour de contrôle envoie le message `décollage` à deux avions. Si l'un des avions est un A320, il possède peut-être une méthode `décollage()` qui lui est propre. Le polymorphisme permet à l'objet `tour de contrôle` de dialoguer avec des avions, quels que soient leurs classes. Le message `autorisation de décollage()` est alors envoyé indistinctement à tous les types d'avions. Grâce au polymorphisme, chaque objet destinataire déclenche le traitement spécifique qui lui est propre. Cette caractéristique fondamentale permet d'éviter la prolifération des aiguillages dans les programmes. De plus, des sous-types pourront être ajoutés dans le futur sans remettre en cause les utilisateurs du sur-type.

1.5.3 L'héritage : une relation, de multiples utilisations

L'héritage offre plusieurs utilisations, en particulier

- Spécialisation par modification de méthode (utilisation intensive du polymorphisme)
- Spécialisation par extension de l'état ou du comportement
- Classification à partir d'un discriminant
- Factorisation par généralisation
- Implémentation ou réalisation d'interface en C++ (voir étude de cas C++).

L'extension de classe par héritage (que l'on retrouve en Java dans le mot clé `extends`) ou bien la spécialisation induisent parfois une évolution des *contraintes* : les règles d'utilisation des objets de la sous-classe ne sont plus compatibles celles de la sur-classe. Dans ce cas, le principe de substitution de Liskov n'est plus respecté. C++ propose alors l'héritage protégé pour éviter toute utilisation abusive.

N Barbara Liskov a énoncé ce principe fondamental dans « Data Abstraction and Hierarchy » SIGPLAN Notices
 O en mai 1988[TC1].
 T
 E « Ce que l'on souhaite ici, c'est quelque chose comme cette propriété de substitution : si pour chaque objet o_1 de type S, il existe un objet o_2 de type T tel que pour tout programme P défini en termes de T, le comportement de P est inchangé lorsque o_1 se substitue à o_2 , alors S est un sous-type de T. »



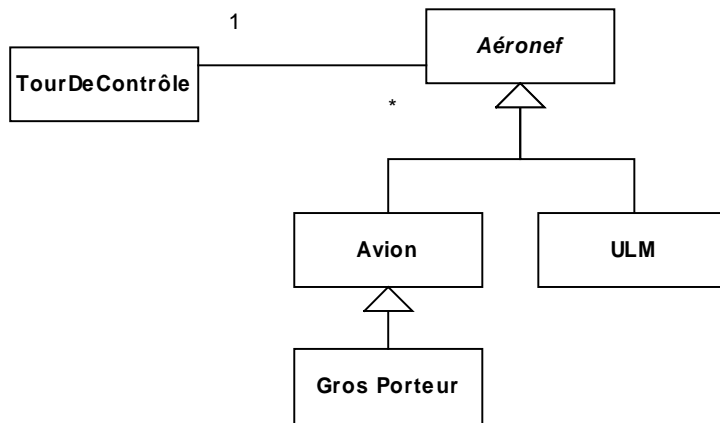


Figure 1-17 : Héritage à problème...

Un Ultra Léger Motorisé (ULM, ne pas confondre avec UML ☺) est un aéronef qui ne peut décoller sans dommage directement après un avion « gros porteur » à cause des turbulences. Les contraintes d'utilisation des classes ne sont pas compatibles : une sous-classe introduit des contraintes supplémentaires. Ainsi, un objet `tour de contrôle` ne peut plus envoyer des messages `décollage` indistinctement de la classe des aéronefs.

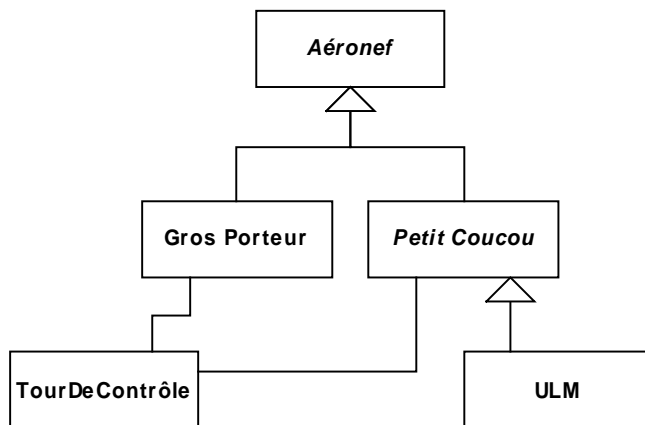


Figure 1-18 : Héritage sans le problème de substitution

1.6 Comparaison des relations entre classes

Les classes sont reliées par association, agrégation / composition et héritage. Ces relations impliquent inéluctablement des couplages. Chaque relation suppose, de façon générale, une durée de vie plus ou moins importante.

Relation	Nature	Durée de vie
Association	Regroupement de liens potentiels entre objets	Généralement temporaire par rapport à la durée de vie des objets ; toutefois, certaines associations naissent avec le système et disparaissent avec lui.
Agrégation, Composition	Relation de type Tout / Partie	Généralement plus longue que l'association
Généralisation	Nature intrinsèque de l'objet	Permanente Couplage fort entre classes

La relation d'héritage entre classes ne correspond pas à une relation entre objets, contrairement aux différentes associations. Autrement dit, une association se traduit par un lien entre deux objets (minimum), l'héritage ne fait intervenir *qu'un* objet de la sous-classe.

CONCEPTS ET NOTATIONS UML	2
1.1 Le concept d'objet	2
1.2 Les interactions entre objets.....	4
1.3 Les classes	6
1.3.1 Classes et interfaces	6
1.3.2 Attributs et opérations	7
1.4 Associations.....	8
1.4.1 Association.....	8
1.4.2 Agrégation.....	9
1.4.3 Composition.....	9
1.4.4 Navigation.....	10
1.5 Généralisation.....	10
1.5.1 Héritage.....	10
1.5.2 Polymorphisme	12
1.5.3 L'héritage : une relation, de multiples utilisations	12
1.6 Comparaison des relations entre classes	13