INF 2005 Programmation orientée objet avec C++

Texte 1

Version du 9 décembre 2014

1. Caractérisation de l'approche orientée objet

La conception par objets est une méthode qui conduit à des architectures logicielles fondées sur des objets que tout système ou sous-système manipulent. Elle est essentiellement caractérisée par l'*encapsulation* des données, par l'*héritage* des attributs et méthodes, et par le *polymorphisme*.

1.1. L'encapsulation des données

L'encapsulation est le mécanisme par lequel le programmeur cache une partie de l'information pour préserver l'intégrité de l'objet. Ce faisant, il établit une séparation entre l'interface (publique) et l'implémentation (invisible). L'encapsulation de données est un arrangement de la structure d'un objet qui permet de réunir des données internes, les méthodes d'utilisation et de manipulation de telles données, ainsi que l'accès à ces données contrôlé par un filtre; la figure 1 illustre ce concept. Ainsi, l'encapsulation facilite la représentation et l'organisation des connaissances et des données, et influe sur l'usage ultérieur de l'objet.

L'encapsulation est aussi appelée *dissimulation*, en ce sens que l'accès à un ensemble défini de structures de données est limité à une liste de fonctions déclarées par le concepteur. Toute fonction qui n'est pas explicitement autorisée à accéder à la structure interne de l'objet provoquera une erreur.

Les types de base des langages informatiques correspondent généralement à des classes encapsulées. Par exemple, le programmeur ne manipule pas la structure interne binaire d'un type flottant, qui est une structure complexe, décomposée en *exposant* et en *mantisse*. Il se contente de manipuler des objets de ce type par l'intermédiaire de méthodes prédéfinies telles que les opérateurs d'affectation, de multiplication, d'addition, de comparaison et d'affichage.

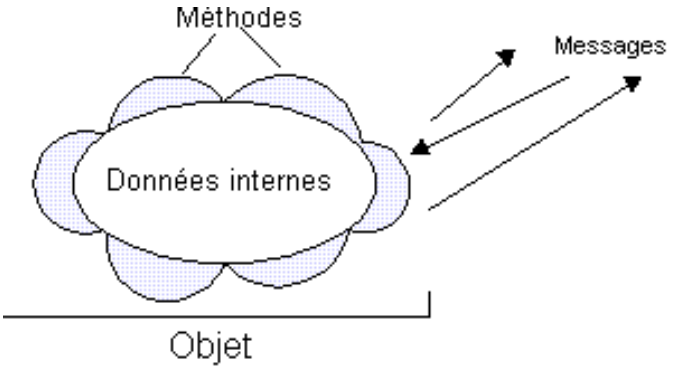

Figure 1 Encapsulation de données.

L’encapsulation a pour principal objectif de masquer l’implémentation des structures au programmeur qui les utilise en fonction de leurs spécifications. Elle est généralement mise en œuvre par une séparation physique entre l’interface qui décrit les spécifications des fonctionnalités et le corps qui en réalise l’implémentation. Ainsi, la modélisation par objets dispose d’un schéma de représentation simple : l’objet en tant qu’encapsulation de données et de traitements, la classe en tant que type abstrait. Plus qu’une simple programmation dirigée par les données, la programmation par objets est une véritable programmation par abstraction de données.

1.2. L’héritage

En général, les systèmes ne naissent pas dans un monde vide. Les nouveaux logiciels sont presque toujours issus de développements antérieurs; la meilleure manière de les créer consiste à procéder par imitation, amélioration et combinaison. Il s'agit là de l'une des préoccupations essentielles de la programmation orientée objet.

Les techniques utilisées jusqu'à maintenant ne sont pas suffisantes, les classes fournissent certes une bonne technique de décomposition modulaire et possède une grande partie des qualités attendues des composantes réutilisables; nous pouvons aussi séparer l'interface de son implémentation grâce au principe de rétention d'information; la généricité leur fournit une flexibilité supplémentaire. Mais il nous faudra aller plus loin pour atteindre de façon optimale les objectifs de réutilisabilité et d'extensibilité.

En effet, pour réaliser des progrès en matière de réutilisabilité et d’extensibilité, il est indispensable d’utiliser des relations entre les classes : une classe peut être une extension, une spécialisation ou une combinaison d’autres classes. Sur ce plan, l’héritage joue les rôles d’outil et de langage qui permettent d’enregistrer et d’utiliser ces relations.

De façon générale, dire qu’une classe `B` hérite d’une classe `A`, c’est dire que les propriétés de `A` sont aussi celles de `B`. La relation d’héritage organise hiérarchiquement les classes en graphes d’héritage. L’héritage est dit *simple* si une classe ne peut directement hériter que d’une superclasse; dans ce cas, le graphe d’héritage est un arbre et le parcours dans la hiérarchie d’héritage est linéaire jusqu’à la racine de l’arbre. L’héritage est *multiple* si une classe peut directement hériter de plusieurs classes de base. Illustrons ces concepts par un exemple.

Supposons que nous désirons créer une librairie graphique pour représenter des figures géométriques. L’une des classes d’une telle librairie pourrait décrire les polygones, avec des opérations telles que le calcul du périmètre, la translation, la rotation, etc. Si nous définissons maintenant une nouvelle classe qui représente les rectangles, nous avons la possibilité de repartir à zéro ou de considérer ces derniers comme des cas particuliers de polygones, avec toutefois des caractéristiques bien spécifiques : le nombre de sommets est 4, les angles sont droits.

Nous pouvons définir une classe `Rectangle` qui hérite de la classe `Polygone`. Dans ce cas, toutes les primitives de `Polygone` sont applicables à `Rectangle` et ne devraient pas se répéter dans cette dernière classe. Un tel processus est transitif : toute classe qui hérite de `Rectangle`, par exemple `Carré`, a aussi les caractéristiques de `Polygone`. De ce fait, le graphe constitue un arbre dont la racine représente la classe la plus générale. Cette dernière est prédéfinie et détient le comportement commun de tous les objets. La figure 2 illustre une hiérarchie de classes graphiques.

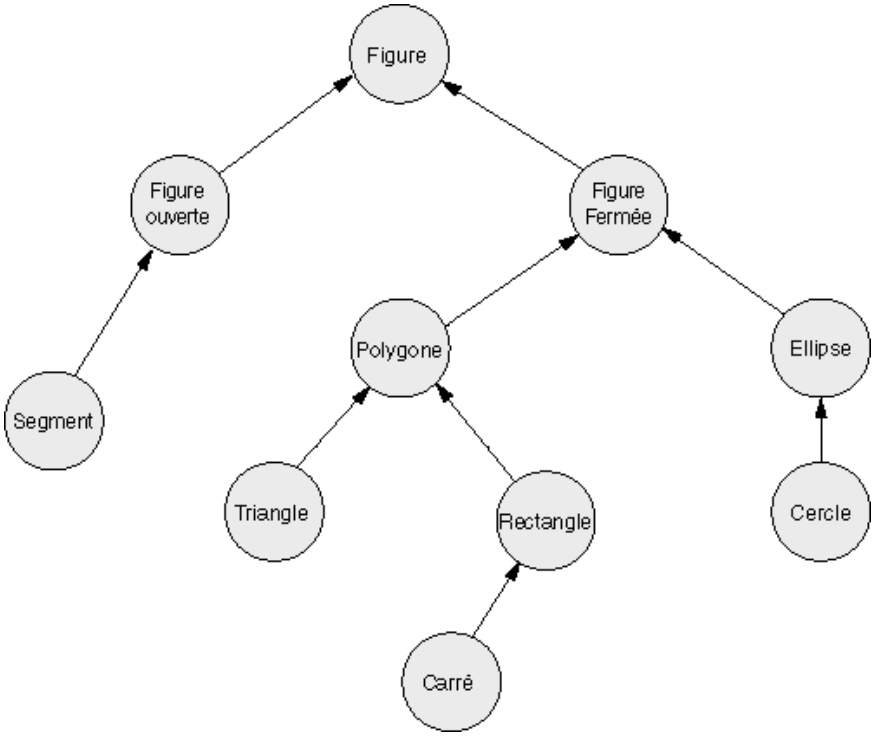

Figure 2 Hiérarchie de classes graphiques.

L'héritage, tel que nous venons de le voir, est simple. Lorsque certaines classes appartenant à des branches différentes de l'arbre partagent certaines propriétés, elles sont reliées à la même superclasse : on parle alors d'héritage multiple.

Dans plusieurs langages, il existe une classe appelée `Object` qui représente l'ensemble de toutes les instances possibles d'une application. Par définition, toutes les classes héritent de la classe `Object` qui constitue alors la racine du graphe d'héritage. Ainsi, en utilisant une telle classe, nous illustrons à la figure 3 un graphe d'héritage multiple. En créant la classe `AlimElectrique`, on regroupe les informations propres à l'alimentation électrique, ce qui permet d'en faire hériter la classe `Téléviseur` d'une part, et la classe `ElectroMénager` d'autre part.

L'avantage de l'héritage multiple est d'accroître la modularité des programmes et d'en faciliter la mise au point et la maintenance. De plus, il permet un gain d'espace

appréciable par la mise en commun d'informations, programmes et données, qu'il faudrait autrement dupliquer dans plusieurs classes. Toutefois, le contenu et l'agencement des classes doivent être développés avec beaucoup de soin. Lorsque le graphe d'héritage devient complexe, son exploitation requiert beaucoup d'expérience.

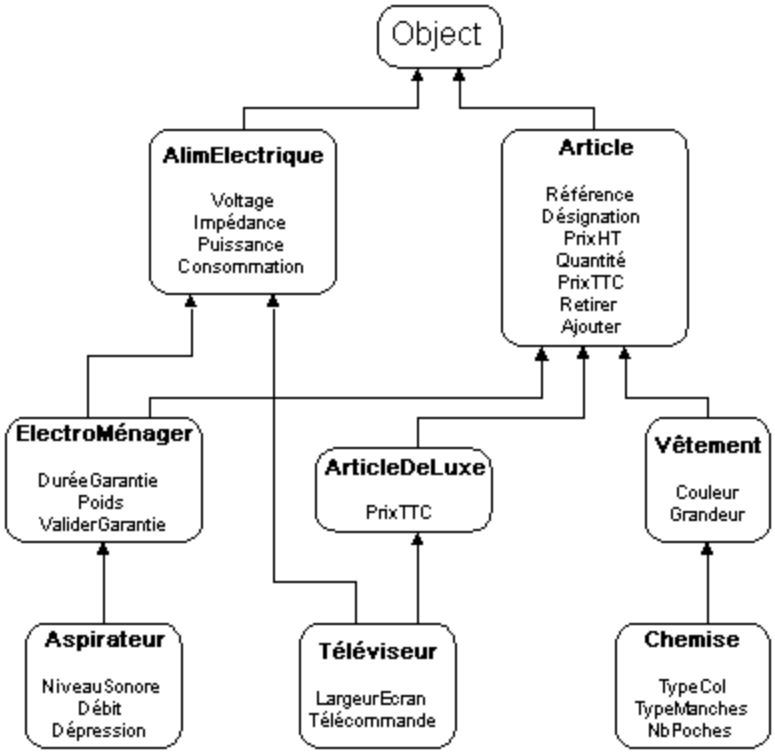

Figure 3 Héritage multiple.

Dans certains cas, lorsqu'une classe `B` a besoin d'une classe `A`, il peut y avoir hésitation sur les relations à utiliser : `B` est-il un héritier de `A` ou son client? L'héritage est approprié si toute occurrence de `B` peut aussi être considérée comme une occurrence de `A`, ce qui donne lieu à la relation « est »; par exemple, un rectangle est un polygone.

Quant à la relation client, elle est appropriée lorsque toute occurrence de `B` possède simplement un ou plusieurs attributs de type `A`, ce qui donne lieu à la relation « *a* ». Une erreur typique consiste à déclarer, par exemple, `Maison` comme héritier de `Porte` et de `Étage`, alors qu'en réalité, une maison possède simplement ces composantes et en est donc un client.

Le mécanisme d'héritage peut entraîner des cas de *surcharge* ou de *conflit*. On parle de surcharge lorsque des opérations de même nom s'appliquent à la fois à des arguments différents de certaines classes et sous-classes. On parle de conflit lorsqu'une classe hérite de deux classes sur lesquelles deux propriétés de même nom sont définies et qu'il y a conflit pour déterminer quelle définition est à prendre en compte. La figure 4 illustre un tel concept.

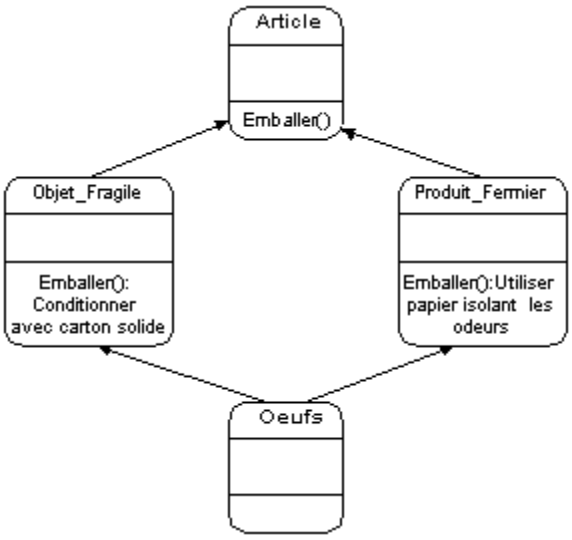

Figure 4 Conflit d'héritage.

Il s'agit d'un problème de partage efficace des connaissances. La classe est considérée comme un réservoir de connaissances à partir duquel il est possible de définir d'autres classes plus spécifiques, complétant les connaissances de leur classe de base. Les connaissances les plus générales sont ainsi mises en commun dans des classes qui sont spécialisées par définition de sous-classes contenant des connaissances de plus en plus spécifiques. Ainsi, une sous-classe constitue une spécialisation de la description de sa *superclasse*.

1.3. Le polymorphisme

Le polymorphisme est la capacité d'obtenir plusieurs réactions à une sollicitation donnée. Autrement dit, plusieurs objets différents peuvent répondre de manière différente au même message. La figure 5 illustre un tel concept. En programmation par objets, le polymorphisme caractérise une entité qui fait référence, au moment de l'exécution, à des occurrences de différentes classes. Par exemple, les opérateurs arithmétiques sont polymorphes par le fait que l'addition, la soustraction peuvent s'appliquer aux entiers, aux réels, voire aux ensembles. Cette caractéristique est la conséquence directe de la communication par envoi de messages.

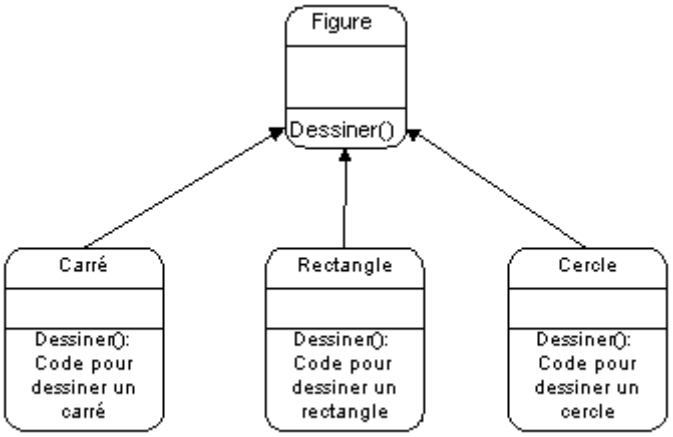

Figure 5 Polymorphisme.

1.4. La communication par envoi de messages

Conceptuellement, le contrôle dans les langages à objets est assuré par des communications entre les objets et non par des appels de procédures. La communication par envoi de messages est la seule structure de contrôle de ces langages. Essentiellement, le protocole d'envoi de messages comporte deux parties : l'*expéditeur* et le *destinataire*. Lors du transfert d'une information, l'expéditeur doit spécifier le nom du destinataire, un sélecteur de nom de message et des arguments. Distinguons aussi deux types de messages : les *requêtes* et les *transmissions*. Les requêtes sont des messages pour lesquels l'expéditeur attend une réponse de la part du destinataire, alors que les transmissions constituent des messages pour lesquels l'objet expéditeur n'attend aucune réponse de l'objet destinataire.

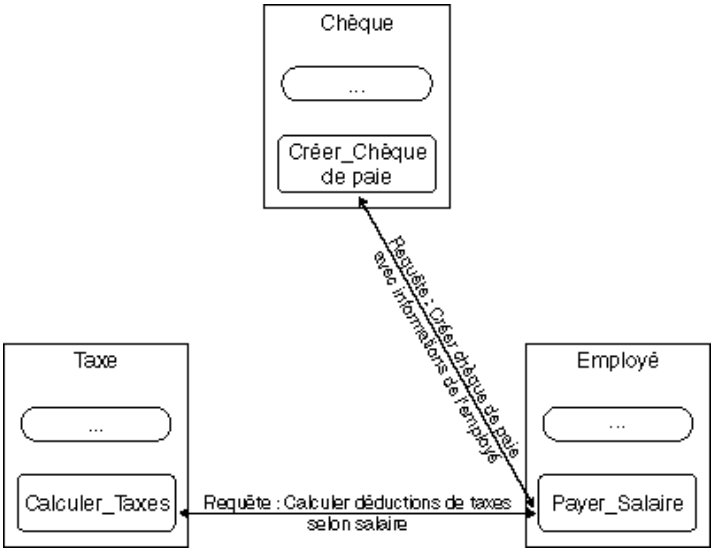

Figure 6 Envoi de messages entre plusieurs classes.

2. Programmation orientée objet

L'approche orientée objet est un concept nouveau qui a rapidement envahi le milieu du développement des logiciels. Elle est présentée comme une technologie qui mène à une meilleure intégration des logiciels et, de ce fait, à une augmentation de la productivité dans ce domaine.

Les techniques de programmation orientée objet permettent au programmeur, d'une part, d'utiliser des éléments de code préalablement développés, comme en ingénierie où l'on peut assembler des modules intégrés possédant des spécifications bien établies; la réutilisation des composants logiciels réduit la complexité du code développé, ainsi que le temps et les efforts nécessaires à son développement. D'autre part, la programmation orientée objet permet aux utilisateurs et aux programmeurs de visualiser les concepts comme une variété d'unités ou d'objets, comme une hiérarchie de composants logiciels ou de structures d'organisation diverses. Cela permet aux programmeurs de représenter facilement les relations entre les composantes, les objets, les tâches à accomplir et les conditions à remplir.

Voyons maintenant les concepts de base, ainsi que les principales caractéristiques de la programmation et des langages orientés objet.

2.1. Le concept d'objet et la conception par objets

Un objet est perçu comme une représentation, un modèle exact et complet d'un objet du monde réel. En programmation orientée objet, il est considéré comme une entité cohérente définie par une structure et un comportement ou comme une entité cohérente rassemblant des données et le code utilisant ces données.

La conception par objets trouve ses fondements dans la réflexion menée autour de la vie du logiciel. D'une part, le développement de logiciels de plus en plus importants nécessite l'utilisation de règles permettant d'assurer une certaine qualité de réalisation. D'autre part, la réalisation d'un logiciel comporte plusieurs étapes, dont le développement ne constitue que la première partie. Elle est suivie dans la plupart des cas de l'étape de maintenance qui consiste à corriger le logiciel, à l'adapter et à le faire évoluer. On estime que cette étape exige plus encore que celle du développement d'un logiciel de qualité.

La conception par objets est issue des réflexions sur la qualité des logiciels, laquelle est assurée si les critères suivants sont respectés :

- *la validité*, soit le fait qu'un logiciel effectue exactement les tâches pour lesquelles il a été conçu;
- *l'extensibilité*, soit la facilité d'adaptation des produits logiciels aux changements de spécifications;

- *la réutilisabilité*, soit la facilité des éléments logiciels à servir à la construction de plusieurs applications différentes;
- *la portabilité*, soit la facilité avec laquelle des produits logiciels peuvent être transférés d'un environnement logiciel ou matériel à un autre;
- *la robustesse*, soit la capacité des systèmes logiciels à réagir de manière appropriée à la présence de conditions anormales;
- *la modularité*, soit une architecture assez flexible pour intégrer des entités indépendantes : les modules.

Voyons plus en détail, dans un premier temps, deux critères assurant des logiciels de qualité, soit la modularité et la réutilisabilité. Nous définirons ensuite les bases générales de la conception par objets, avant d'étudier la terminologie associée à cette approche.

2.1.1 La modularité

Les critères énoncés ci-dessus influent sur la façon de concevoir un logiciel, en particulier sur l'architecture logicielle. En effet, certains de ces critères ne sont pas respectés lorsque l'architecture d'un logiciel est monolithique. Dans ces conditions, le moindre changement d'une spécification peut avoir des répercussions très importantes sur le logiciel, imposant une lourde charge de travail pour effectuer l'entretien et les mises à jour. Pour la conception des logiciels, on adopte généralement une architecture assez flexible pour éviter un tel problème, soit une architecture intégrant des entités indépendantes : les modules.

L'intérêt de ce type de conception est de concentrer les connaissances liées à une entité logique à l'intérieur d'un module, qui est le seul habilité à exploiter ces connaissances. L'une des conséquences immédiates est que, lorsqu'une mise à jour doit être effectuée sur une entité logique, celle-ci ne touche qu'un seul module, ce qui confine le travail de maintenance.
Il existe deux grandes familles de méthodes de conception de modules.

- Les *méthodes descendantes* qui procèdent par la décomposition du problème. Le problème est ainsi divisé en un certain nombre de sous-problèmes, chacun étant de complexité moindre. Cette division est ensuite appliquée aux sous-problèmes, et ainsi de suite, jusqu'à ce que chacun des sous-problèmes ne puisse plus être décomposé.
- Les *méthodes ascendantes* qui procèdent par composition de briques logicielles simples, pour obtenir des systèmes complets. C'est en particulier le cas des bibliothèques de sous-programmes, disponibles avec tous les systèmes, langages et environnements.

Les deux méthodes ne s'opposent pas automatiquement et sont souvent utilisées en parallèle lors de la conception d'un logiciel. Il faut cependant noter que l'approche descendante ne favorise pas toujours la réutilisabilité des modules produits.

Outre la démarche de conception de modules, précisons quelques critères de qualité à respecter lors de la définition des modules.

- *Compréhensibilité modulaire.* Les modules doivent être clairs et organisés de manière compréhensible dans le système. Ceci implique que les modules doivent communiquer avec peu de modules, ce qui permet de les « situer » plus facilement. De même, l'enchaînement des différents modules doit être logique; par exemple, on ne doit pas utiliser plusieurs fois de suite un module pour produire une action atomique.

- *Continuité modulaire*. Ce critère est respecté si une modification dans les spécifications n'entraîne qu'un nombre limité de modifications au sein d'un petit nombre de modules, sans remettre en cause les relations qui les lient.

- *Protection modulaire.* Ce critère signifie que toute action lancée dans un module doit être confinée à ce module, et éventuellement à un nombre restreint de modules. Ce critère ne permet pas de corriger les erreurs introduites, mais de confiner, pourvu que cela soit possible, les erreurs dans les modules où elles sont apparues. Ces notions, assez intuitives et découlant des réflexions sur la vie des logiciels, doivent être prises en compte lors de la définition et de la maintenance des modules, même si elles ne sont pas accompagnées d'une méthodologie précise permettant d'y arriver; elles assurent la qualité globale d'un logiciel.

À partir des critères exposés ci-dessus, quelques principes de définition ont été retenus pour la conception des modules.

- *Qu'il y ait peu d'interfaces*. Le but est de se restreindre à un nombre limité d'actions bien définies, ce qui supprime une part des erreurs liées à l'utilisation des modules. En somme, chaque module devrait communiquer avec un minimum de modules.

- *Que les communications limitées entre les modules, réalisées par leur interface, soient limitées quantitativement*. Ceci est une conséquence de la modularité; ce critère est d'autant mieux respecté, lorsque les modules jouent bien leur rôle. Si les échanges sont trop nombreux ou importants, la notion même de module devient floue, limitant l'intérêt de cette technique.

- Que les interfaces soient explicites de façon que les communications entre les modules ressortent explicitement.

- *Que le masquage de l'information soit assuré*. Toutes les informations contenues dans un module doivent être déclarées « privées » au module, à l'exception de celles explicitement définies « publiques », notions que nous allons voir plus loin.

Les communications autorisées sont ainsi celles explicitement définies dans l'interface du module, par les services qu'il propose.

- Que les modules définis lors de la conception correspondent à des unités modulaires syntaxiques liées au langage de programmation. Autrement dit, le module spécifié ne doit pas s'adapter au langage de programmation, mais au contraire, le langage de programmation doit proposer une structure permettant d'implanter le module tel qu'il a été spécifié. En effet, les variables globales peuvent être utilisées et modifiées par n'importe quelle composante d'un programme, ce qui complique d'autant la maintenance de telles variables.

2.1.2 La réutilisabilité

La réutilisabilité n'est pas un concept nouveau en informatique et remonte aux débuts de l'informatisation. En effet, les types de données à stocker sont toujours construits autour des mêmes bases (tables, listes, ensembles) et la plupart des traitements comportent des actions atomiques, telles que l'insertion, la recherche, le tri, etc., qui sont des problèmes résolus en informatique.

Il existe une documentation assez abondante décrivant les solutions optimales pour chacun de ces problèmes. La résolution des problèmes actuels passe par les solutions de chacun de ces problèmes de base. Les bibliothèques (systèmes, mathématiques, etc.) sont de bons exemples de réutilisabilité et sont couramment utilisées par les programmeurs; elles ont cependant des limites. En effet, les fonctions qu'elles comportent ne sont pas capables de s'adapter aux changements de types de données ou d'implantation. Dans un tel cas, la solution est de fournir une multitude de fonctions, chacune étant adaptée à un cas particulier, ou d'écrire une fonction prenant en compte toutes les possibilités. Dans un cas comme dans l'autre, c'est insatisfaisant. C'est pourquoi la conception par objets formalise un peu plus cette notion de réutilisabilité et propose de nouvelles techniques pour y arriver.

Dans la section portant sur la notion de module, nous avons insisté sur les avantages de la conception modulaire, mais nous avons peu traité de la conception même d'un module. Convenons d'abord qu'un « bon » module est réutilisable, c'est-à-dire conçu dans l'optique d'être placé dans une bibliothèque à des fins de réutilisation. Afin de « marier » efficacement modularité et réutilisabilité, quelques principes ont été définis pour la conception de « bons » modules.

- *Un module doit pouvoir manipuler des données de plusieurs types différents*. Un module de listes, par exemple, doit pouvoir manipuler aussi bien des entiers que des types composés.

- *Un module doit pouvoir s'adapter aux différentes structures de données manipulées et dotées de méthodes spécifiques*. Il devra ainsi, par exemple, pouvoir rechercher de la même manière une information contenue dans un tableau, une liste, un fichier, etc.

- *Un module doit pouvoir offrir des opérations aux clients qui l'utilisent sans que ceux-ci en connaissent l'implantation.* Ceci est une conséquence directe du masquage de l'information préconisé. C'est, en outre, une condition essentielle pour développer de grands systèmes; les clients d'un module sont ainsi protégés de tout changement de spécifications se rapportant à un module.

- *Les opérations communes à un groupe de modules doivent pouvoir être factorisées dans un même module*, c'est-à-dire une manière de faire apparaître le minimum de fois chaque occurrence d'événement de base. Ainsi, les modules effectuant du stockage de données, telles les listes, les tables, etc., doivent être dotés d'opérations qui utilisent un même nom permettant d'accéder à des éléments, d'effectuer un parcours, de tester la présence d'éléments. Ceci peut permettre, entre autres, de définir des algorithmes communs, telle la recherche, quelle que soit la structure utilisée pour stocker les données.

2.1.3 De nouvelles techniques

Pour faire cohabiter les principes de la modularité et de la réutilisabilité, la notion de paquetage est utile; elle a été introduite par des langages tels que Ada ou Modula-2. Un paquetage correspond à un regroupement, au sein d'un même module, d'une structure de données et des opérations qui lui sont propres. Ceci satisfait en particulier les critères de modularité, en isolant chaque entité d'un système, ce qui la rend plus facile à mettre à jour et à utiliser. En ce qui concerne les critères de réutilisabilité, il est possible d'aller encore un peu plus loin. Voyons ici de nouvelles notions qui apparaissent avec le paquetage et qui vont permettre de franchir ce pas.

- La *surcharge.* Cette notion prévoit que des opérations appartenant à des modules différents peuvent être associées au même nom. Les opérations ne sont plus indépendantes, mais elles prennent leur signification contextuellement, en fonction du cadre dans lequel elles sont utilisées. Parmi ces opérations se trouvent les fonctions, mais également les opérateurs; par exemple, en définissant une fonction « insérer » dans chaque module de stockage, cela permettra d'écrire de manière uniforme : « insérer(élément, contenant) », quel que soit le type de conteneur (liste, tableau, fichier, etc.).

- La *généricité.* Cette notion permet de définir des modules paramétrés par le type de données qu'ils manipulent. Un module générique n'est alors pas directement utilisable; c'est plutôt un modèle de module qui sera « instancié » par les types de paramètres qu'il accepte. Cette notion est très intéressante, car elle va permettre la définition de méthodes (façons de travailler) plutôt que la définition de fonctions (plus formelles). Ces définitions et ces nouveaux outils vont permettre de définir de nouvelles manières de concevoir des systèmes informatiques. D'une manière générale, c'est une facilité accordée aux auteurs du module « fournisseur ». La généricité rend possible l'écriture d'un texte

fournisseur unique pour les diverses implémentations d'un même concept appliqué à différents types d'objets.

2.2. La conception par objets

Afin d'établir de façon stable et robuste l'architecture d'un système, il semble maintenant plus logique de s'organiser autour des données manipulées, les objets. En effet, les données étant, de par leur nature, plus stables que les traitements, la conception en est simplifiée. De plus, il apparaît que la conception par traitements ne favorise pas l'utilisation des principes de qualité, tels que la modularité ou la réutilisabilité. Il reste maintenant à éclaircir les fondements mêmes de la conception par objets.

Dans cette optique, voici une première définition énoncée par Meyer[1] : « La conception par objets est la méthode qui conduit à des architectures logicielles fondées sur les OBJETS que tout système ou sous-système manipule. » Il ajoutait : « Ne commencez pas par demander ce que fait le système, demandez À QUOI il le fait! »

La spécification d'un système va donc maintenant s'axer principalement sur la détermination des objets manipulés. Une fois cette étape franchie, le concepteur n'aura plus qu'à définir les fonctions de haut niveau qui s'appuient sur les objets et les familles d'objets définis. C'est cette approche, préconisant de considérer d'abord les objets (c'est-à-dire les données) avant l'objectif premier du système à concevoir, qui permet d'appliquer les principes de réutilisabilité et d'extensibilité. Reste maintenant à décrire la structure d'un objet, son comportement et son exploitation, ce qui nous conduira à la notion de classe.

2.2.1 La structure d'un objet

Dans la réalité, physique ou abstraite, les objets sont présents partout. Le modèle du monde est en effet constitué d'objets : étudiants, employés, avions, chèques, etc. Les objets logiciels reflètent simplement ces objets externes. Dans un système graphique, les objets sont les points, les lignes, les polygones, les cercles, etc.; dans un système de paie, les objets sont les employés, les chèques, les échelles de salaire et ainsi de suite. La conception par objets consiste essentiellement à manipuler ces objets concrets de la vie de tous les jours. Toutefois, les opérations à effectuer sur chacun requièrent un minimum de connaissances sur leur structure.

Un objet se compose essentiellement de deux parties : une partie statique qui regroupe un ensemble de données touchant l'objet et une partie dynamique regroupant l'ensemble des procédures et fonctions manipulant ces données. Les données

[1] Meyer, B. (1991). *Conception et programmation par objets* (2e éd.). Informatique intelligence artificielle. Paris, France : InterÉditions.

constituent les attributs de l'objet, alors que les procédures sont des méthodes. De plus, l'objet est muni d'une interface qui spécifie les interactions qu'il peut avoir avec le monde extérieur.

La figure 7 illustre, d'une façon générale, le concept d'objet. Nous pouvons en déduire que toute communication avec un objet doit absolument se faire en invoquant des méthodes correspondantes, par l'intermédiaire de son interface. Ainsi, l'objet cache au monde extérieur les détails de son implémentation, ce qu'on appelle dans le jargon du domaine l'abstraction des données. Alors, toute modification respectant les spécifications de son interface aura des conséquences limitées sur cet objet et ne touchera en rien le système global dont il fait partie.

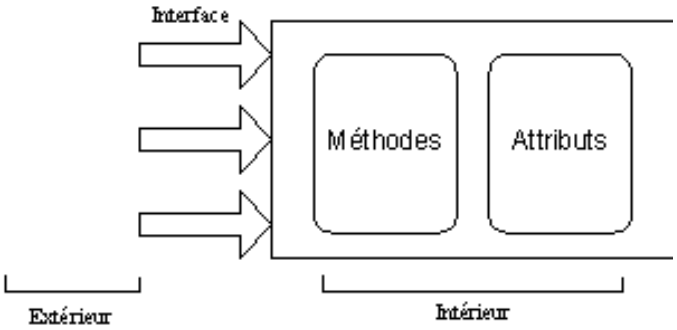

Figure 7 Concept d'objet.

2.2.2 Le comportement d'un objet

Tout objet a un certain comportement qui dépend du milieu dans lequel il évolue. En programmation orientée objet, les applications sont essentiellement composées d'un ensemble d'objets, chacun détenant les clefs de son comportement. Ces objets doivent trouver une manière de communiquer entre eux : l'envoi de messages est le mode privilégié d'échange entre les objets. Un message est une structure qu'on peut formellement décrire comme suit :

```
MESSAGE = <objet destinataire, méthode à exécuter, arguments>
```

Les arguments sont les valeurs passées en paramètres à la méthode activée.

Par ailleurs, le processus d'envoi de messages aboutit à l'activation de certaines méthodes selon un protocole prédéfini; cela fait référence à des notions *de fonctions* et de *services.*

Une fonction est une activité (ensemble de traitements) qui conduit à la satisfaction d'un besoin général ou à l'obtention d'un résultat particulier. Un service est un comportement spécifique qu'adopte un objet à la demande d'un autre objet. La figure 8 illustre une sollicitation de services par laquelle un objet demande à l'objet *Employé* d'exécuter la méthode *Payer_salaire*. L'appel d'une méthode est une requête, c'est-à-dire un message de l'appelant demandant l'exécution d'une certaine

action. Les relations entre les objets s'établissent par l'application de certains principes de coopération, selon lesquels un objet sous-traite une partie de son activité en sollicitant des services à d'autres objets.

Une fois construit, un objet est vivant et demeure en attente permanente de messages. À la réception d'un message, cet objet devient actif et effectue le traitement spécifié par le message reçu, en manipulant son espace de données ou en activant d'autres objets; il s'occupe alors de la manière dont les opérations sont exécutées. La programmation est dirigée par les données. Autrement dit, pour traiter une application, le programmeur définit les types d'objets appropriés, avec leurs comportements spécifiques, chaque entité manipulée dans le programme étant un représentant d'un de ces types.

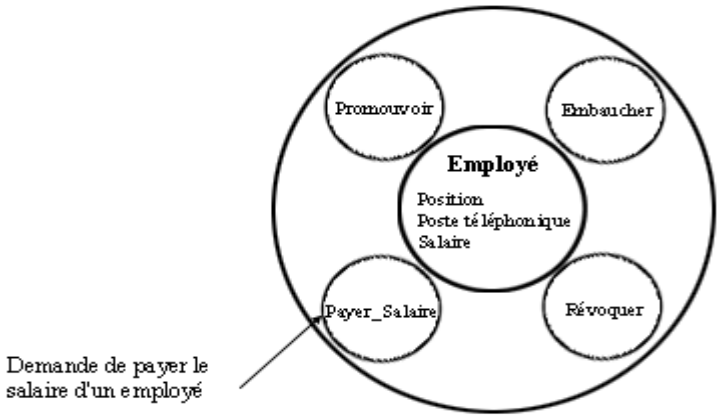

Figure 8 Sollicitation de services.

3. Langages à objets

La programmation orientée objet permet de structurer l'univers des applications en fonction d'objets plutôt qu'en fonction de procédures. Dans ce contexte, les langages à objets, regroupant les langages de programmation qui intègrent la notion d'héritage à l'abstraction de données, sont largement utilisés. Cette section 3 présente les catégories de ces langages, en mettant l'accent sur les langages orientés objet.

3.1. Classification

Il existe plusieurs catégories de langages à objets : les langages basés objet, les langages de classes, les langages acteurs et les langages dits orientés objet.

Un langage de programmation est dit basé objet s'il autorise la modélisation directe d'objets du monde réel, tels des pièces mécaniques, des automobiles, des personnes ou des comptes en banque, l'objet étant défini comme un ensemble d'opérations et un état qui mémorise les derniers effets de ces opérations. Les objets de tels langages ne contiennent pas nécessairement de classes d'appartenance, ni d'héritage : ils sont

appelés paquetages (*packages*) et n'appartiennent à aucune classe. Le langage Ada constitue un exemple de langage basé objet.

Un langage de classes est un langage qui supporte la notion de classe, laquelle permet de réunir des objets et de les assimiler à un modèle. Les objets de ces classes peuvent être transmis en paramètres, affectés à des variables et organisés en structures. Toutefois, il n'existe pas de relation hiérarchique entre les données des différents niveaux. Autrement dit, les relations d'héritage n'existent pas dans ces langages, qui constituent en fait des langages de classes. Le langage CLU, créé au Massachusetts Institute of Technology (MIT), en est un exemple.

Un langage est orienté objet si, à la fois, il est basé objet, supporte la notion de classe et autorise l'héritage. D'où la relation suivante :

```
Orienté objet = basé objet + classe + héritage
```

Quant aux langages de programmation orientée objet concurrente, ils mettent en jeu des processus qui peuvent s'apparenter à des acteurs.

Dans cette famille de langages, chaque objet, appelé acteur, constitue un processus qui s'exécute de façon autonome, émettant et recevant des messages d'autres acteurs. Un acteur peut en créer un autre par simple reproduction ou copie de lui-même. La copie engendrée est alors soit rigoureusement identique, soit différente (spécialisée par adjonction de nouvelles caractéristiques). Un tel mécanisme s'apparente à l'instanciation des langages orientés objet. L'organisation des langages de programmation peut se résumer comme l'illustre la figure 9.

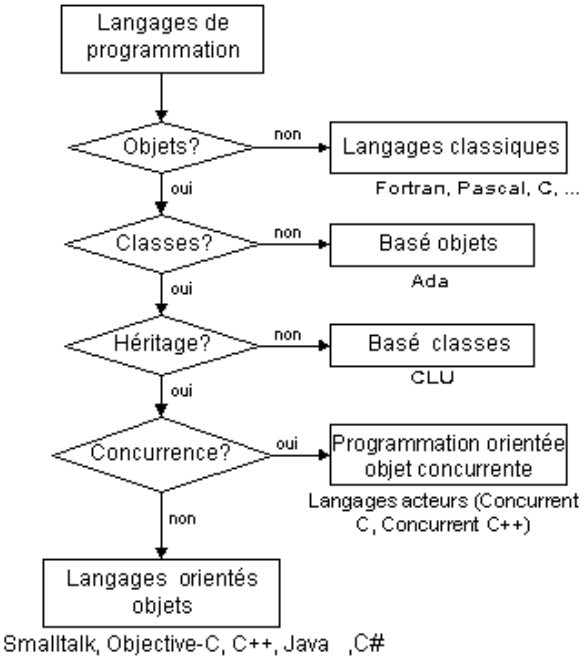

Figure 9 Classification des langages de programmation.

3.2. Les langages orientés objet

Présentons maintenant les langages orientés objet les plus connus : Smalltalk, Objective-C, C++, Java et C# (prononcez *C Sharp*).

3.2.1 Smalltalk

Smalltalk est à l'origine de la vague des langages à objets et constitue un environnement complet, interactif et orienté objet, qui fournit des fenêtres, des graphismes, des éditeurs et des facilités de mise au point. En tant que langage de programmation orienté objet, Smalltalk permet à l'utilisateur de classer les informations en une hiérarchie de caractéristiques reliées entre elles comme des classes ou des sous-classes. Les problèmes peuvent être découpés en sous-problèmes plus accessibles. Ces sous-problèmes peuvent être prototypés sans une longue analyse préalable.

Smalltalk ne ressemble à aucun langage de programmation conventionnel. Avant d'être efficace et de maîtriser la puissance de ce langage, un programmeur doit posséder des connaissances approfondies des classes et des méthodes incluses dans sa librairie. Du point de vue conceptuel, Smalltalk introduit la notion de métaclasse, accessible par l'utilisateur de manière naturelle et permettant une programmation plus facile.

3.2.2 Objective-C

Objective-C est un langage hybride qui greffe les possibilités de Smalltalk sur une syntaxe C. De ce fait, la compréhension de Smalltalk procure une bonne base pour l'aborder. Le type de données objets est ajouté aux types reconnus par C. Une nouvelle opération, l'expression de message, gère les liens à l'exécution, de sorte qu'il n'existe aucune distinction entre les classes et les objets lors de la compilation. La puissance d'Objective-C réside dans les ensembles de classes prédéfinies fournies avec le compilateur. Ainsi, un programmeur maîtrise un tel langage en assimilant parfaitement l'ensemble de ces classes prêtes à être utilisées. On se sert d'Ojective-C comme langage de développement de base sur la station de travail *NeXT*; il s'adapte parfaitement bien à la réalisation des interfaces utilisateurs.

3.2.3 C++

C++ est conçu à partir du langage C et en constitue une extension; c'est un nouveau langage, homogène, gardant dans une très large mesure la compatibilité avec C, de telle sorte qu'un programme C++ puisse utiliser les librairies existantes du compilateur C. De plus, la sémantique et la syntaxe de base sont les mêmes pour les deux langages, ce qui rend possible la combinaison des fonctions de C et de C++. Ce langage possède les mêmes facilités que C, en ce qui concerne l'interfaçage direct avec le matériel, et il améliore grandement la définition des interfaces de communication entre les différents modules des programmes.

Le concept de classe proposé par C++ est similaire à celui qui a été développé dans Smalltalk. La programmation orientée objet est centrée sur les données manipulées plutôt que sur les procédures qui effectuent les manipulations. La principale tâche d'un programmeur C++ est le découpage d'un problème en types de données imbriquées (classes et relations entre classes), ainsi que la définition des propriétés et du comportement de ces classes (méthodes). Les variables de classes et de classes dérivées correspondent aux entités physiques ou logiques qui régissent le problème réel.

C++ supporte aussi les notions linguistiques de classe (encapsulation), d'héritage, de surdéfinition ou surcharge d'opérateurs. C'est un langage riche et puissant qui facilite une approche structurée de la programmation. Les mécanismes de protection des données et des méthodes y sont très perfectionnés et la possibilité de surcharger les opérateurs, au même titre que les fonctions, en constitue un point fort. D'ailleurs, C++ a été utilisé dans de nombreuses applications dont Pi, un débogueur interactif avec multifenêtrage, l'environnement interactif InterWiews sous le système X Windows, l'interface utilisateur générique Open Dialogue conçue par Apollo, et une implantation du standard graphique GKS.

3.2.4 Java

Java est un nouveau langage orienté objet et dynamique, avec des éléments de C, C++ et d'autres langages. Il est considéré comme une version améliorée de C++ et comporte des librairies adaptées à l'environnement Internet. Selon Sun Microsystems, Java est conçu pour être un langage simple, orienté objet, distribué, interprété, fiable, robuste, sûr, indépendant de l'architecture matérielle, portable, performant et dynamique.

Dans le même ordre d'idées, Java est aussi conçu pour distribuer du contenu exécutable sur les réseaux et, dans ce contexte, offre plusieurs véritables possibilités : animer les pages web et enrichir la présentation des informations à l'écran sous la forme de séquences animées et d'applications interactives.

3.2.5 Le langage C#

Le langage C# (prononcez *C Sharp*) a été conçu par Microsoft pour en faire le langage de prédilection de sa plateforme .NET. Il peut être vu comme une réponse de Microsoft au langage Java de Sun, et a été annoncé au public au milieu de l'année 2000. Le langage C# s'inspire directement du C++, de Java et de Visual Basic.

De façon particulière, le langage C# utilise un « ramasseur de miettes » (*garbage collector*) pour libérer le programmeur de la gestion de la mémoire; les variables sont automatiquement initialisées et les constructions qui peuvent introduire des erreurs, lorsqu'elles sont mal utilisées, sont défavorisées. En outre, un programme écrit en C# est d'abord converti en un langage intermédiaire, analogue à la machine virtuelle de Java, et ce langage intermédiaire est commun aux différents langages de la plateforme .NET. Microsoft a soumis le langage à l'organisation ECMA pour en faire un standard

reconnu. Notons que la plateforme .NET utilise d'autres langages de programmation tels que VISUAL BASIC.NET, ASP.NET et C++.NET.

Le langage C# est un langage orienté composant logiciel conçu pour la programmation du web. Il reprend la plupart des concepts de SmallTalk et de Java en les intégrant dans une syntaxe héritée du C++. Le langage C# corrige les défauts du C++ tout en l'étendant.

4. Abstraction et classe

4.1. L'abstraction

Les deux principaux paradigmes (principes) de la programmation objet sont l'*abstraction* et l'*encapsulation*.

- L'*abstraction*, c'est la capacité d'ignorer les détails d'un élément pour en avoir une vision globale. L'abstraction de code (ou abstraction procédurale) correspond donc à ignorer les détails d'une portion de code pour la voir comme un tout : il s'agit des procédures et des fonctions! L'abstraction de données consiste à regrouper des données, ce qui aboutit aux types complexes de données (les structures en C).

Le modèle objet est une synthèse de ces deux types d'abstraction : un objet est à la fois une abstraction de données et une abstraction de code.

- L'*encapsulation*, c'est le masquage de certains éléments au sein d'une entité (un « module », ou un objet). Ils ne sont plus visibles (ou accessibles) à l'extérieur de l'entité. La mise en œuvre rigoureuse de ce concept lors de la conception favorise la réutilisabilité et la maintenance.

Dans le modèle objet, un objet n'est visible de l'extérieur que par son interface.

Le grand mérite de l'encapsulation est que, vu de l'extérieur, un objet se caractérise uniquement par les spécifications de ses méthodes, la manière dont sont réellement implantées les données étant sans importance. On décrit souvent une telle situation en disant qu'elle réalise une « abstraction des données» (ce qui exprime bien que les détails concrets d'implémentation sont cachés).

4.2. La notion de classe

Comme nous l'avons déjà mentionné, un objet est caractérisé par sa structure et son comportement. Toutefois, nous pouvons regrouper tous les objets ayant la même structure et le même comportement dans une catégorie spécifique d'objets, appelée *classe*. Voyons comment représenter et déterminer une classe.

4.2.1 La représentation d'une classe

Une classe est la description d'un ensemble d'objets ayant la même structure et le même comportement. Elle constitue une entité génératrice d'une famille d'objets dont elle définit la structure et le comportement par les propriétés relationnelles (ses attributs) et fonctionnelles (ses méthodes).

Autrement dit, chaque classe possède une double composante : une *composante statique* (les données) constituée de champs qui caractérisent l'état des objets pendant l'exécution du programme, et une *composante dynamique* (les procédures); celles-ci, aussi appelées méthodes, traduisent le comportement des objets appartenant à la classe en question et en manipulent les champs.

Une classe possède une interface et un corps. L'interface d'une classe constitue la partie accessible à son utilisateur. Plus précisément, elle fournit la spécification de la classe, en indiquant ses propriétés, en précisant l'ensemble de ses instances et en décrivant les moyens permettant d'utiliser les services qu'offre cette classe.

Quant au corps de la classe, il décrit son implémentation, c'est-à-dire la manière dont les traitements sont effectués. Seul le programmeur d'une classe, qui voudrait par exemple modifier la valeur d'un attribut, peut accéder à la représentation interne de ses propriétés.

Pour illustrer notre propos, concevons un système permettant de gérer le stock d'un magasin. Dans ce cas, nous pouvons créer une classe d'articles avec, comme attributs, les champs suivants :

- `Référence`, qui désigne le numéro de référence de cet article;
- `Désignation`, représenté par un texte qui décrit l'article en question;
- `PrixHT`, indiquant le prix unitaire de l'article;
- `Quantité`, qui désigne le nombre d'articles disponibles dans le stock.

En outre, la définition de la classe doit être complétée par une liste de méthodes qui permettent de manipuler les champs mentionnés plus haut. Chaque méthode est définie par un nom, souvent appelé sélecteur, une liste de paramètres et un corps de fonction. Pour notre exemple, nous pouvons définir les méthodes suivantes :

- `CalculerPrixTCC`, qui calcule le prix d'un article toutes taxes comprises, en multipliant le prix avant taxe par le taux de taxation;
- `CalculerPrixtransport`, qui calcule le prix de transport de l'article en considérant que ce coût équivaut à un certain pourcentage, par exemple 5 % du coût avant taxe de l'article;

- `Retirer`, qui permet de retirer des articles du stock en soustrayant la valeur de son paramètre au champ Quantité;
- `Ajouter`, qui permet d'ajouter des articles au stock.

Voici un exemple d'une implémentation possible de la classe `Article`.

```
Classe

Article


Champs

Référence
Désignation
Quantité



Méthodes
CalculerPrixTCC() :
   retourner (1.15 * PrixHT)
CalculerPrixTransport() :
   retourner (0.05 * PrixHT)
Retirer (q) :
   Quantité = Quantité – q
Ajouter (q) :
   Quantité = Quantité + q
```

4.2.2 La détermination des classes

Les programmeurs habitués à utiliser l'approche fonctionnelle doivent encore se demander comment trouver, déterminer les classes. La réponse à une telle question est simple : le talent et l'expérience sont des atouts fondamentaux pour déterminer des classes. Toutefois, il existe quelques principes généraux qui peuvent nous éclairer sur la façon de concevoir de nouvelles classes.

Ainsi, lorsqu'un ensemble de classes est proposé pour résoudre un problème, nous pouvons les examiner selon des principes et critères de modularité. De cette analyse, nous pouvons déduire les classes qui peuvent être réutilisées pour nos besoins spécifiques. En effet, un bon environnement à objets offre un certain nombre de classes prédéfinies qui implémentent des abstractions importantes, et qu'on peut aller chercher pour éventuellement les réutiliser. C'est le cas des listes, de certains fichiers, des piles, des files, etc. Cela nous amène à la conclusion suivante : plus les techniques à objets se développent, plus le nombre et le niveau d'abstraction des composantes réutilisables augmentent. Ainsi, la réutilisation de certaines classes, dites classes de base, permet de résoudre en partie le problème de détermination de nouvelles classes. La question à se

poser maintenant est la suivante : « Comment créer des classes à partir des spécifications du système à concevoir? »

Lorsque l'on conçoit un système avec des objets, une bonne façon de trouver les classes consiste à souligner les noms, lors de la lecture du document de spécification; pour ce qui est des méthodes fonctionnelles, on souligne surtout les verbes.

Par exemple, la phrase « Le radar doit contrôler la position et la vitesse des automobiles » conduirait un concepteur par objets à détecter deux classes : `Radar` et `Automobile`. Cette technique, d'ailleurs simpliste, ne donne que des résultats approximatifs. En effet, si elle est appliquée à la lettre, elle est susceptible de fournir trop de classes candidates. Dans notre exemple, il n'est pas évident que position et vitesse doivent servir à déterminer des classes.

La création de classes inutiles est une erreur typique de débutant. Il n'existe toutefois pas de méthode sûre pour éviter de telles erreurs. Le fil conducteur demeure la théorie des abstractions de données : une notion ne doit être établie comme classe que si elle décrit un ensemble d'objets caractérisés par des opérations intéressantes et par des propriétés significatives.

Ainsi, dans l'exemple précédent, la position des automobiles peut-elle être considérée comme une classe? Oui et non. S'il n'existe pas d'opération spécifique sur les positions, il vaut mieux considérer `Position` comme un attribut de la classe `Automobile`. Toutefois, si une position est une entité significative, avec des opérations qui lui sont associées (distance d'une autre position, erreur de mesure, conversion à un autre système de coordonnées), alors elle peut être définie comme une classe. Il en est de même pour la vitesse : si la vitesse d'une automobile a des propriétés utiles propres, elle peut être définie par une classe spécifique.

Attention, ne confondez pas objet et classe!

À la lumière de ce qui vient d'être présenté, il existe une différence fondamentale entre classe et objet : les objets existent seulement pendant l'exécution d'un programme, alors que les classes sont des descriptions purement statiques d'ensembles possibles d'objets. Autrement dit, à l'exécution, il n'existe que des objets; mais, dans le programme, nous manipulons des classes.

4.2.3 L'instanciation

Le mécanisme par lequel l'exécution d'un programme produit un objet à partir d'une classe constitue l'instanciation. Par cette dernière, la définition d'une classe sert de modèle à la construction de ses représentants physiques, appelés instances. Une instance constitue un objet particulier créé en respectant les plans de construction de sa classe; cette dernière définit la structure et le comportement de l'instance. Par ailleurs, l'état d'une instance est caractérisé par les valeurs de ses attributs. De plus, un attribut

(ou une méthode) associé à une classe sera défini pour toutes les instances de cette classe. La figure 10 illustre la création de deux instances de la classe *Article*.

Deux instances d'une même classe se distinguent par les valeurs de leurs attributs. Lorsque toutes les instances d'une classe possèdent les mêmes champs, ces derniers prennent des valeurs différentes, correspondant à la nature particulière de chacune de ces instances. Cela est illustré à la figure 10 où l'une des instances représente l'ensemble des pantalons et l'autre, l'ensemble des téléviseurs. Nous nous rendons compte que la structure de données décrite par les champs est dupliquée pour former la structure de chaque instance. Une telle structure est remplie avec les valeurs propres de l'objet représenté par chaque instance.

Si aucune valeur n'est allouée à un attribut lors de la création d'une instance, cet attribut prendra comme valeur pour cette instance sa valeur par défaut. Si, par contre, une valeur initiale a été affectée à un attribut lors de la création d'une instance, l'attribut en question conservera cette valeur jusqu'à ce qu'une autre valeur lui soit affectée. Enfin, une instance ne peut exister sans sa classe d'appartenance, qu'elle reconnaît grâce à la relation d'instanciation (indiquée par *instance de* sur la figure 10).

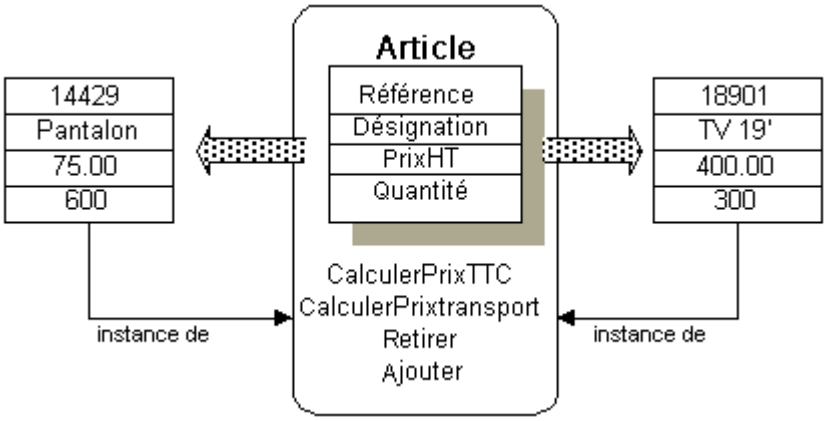

Figure 10 Création de deux instances de la classe `Article`.

5. Contrôle d'accès

Dans la définition d'une classe, nous avons vu que les accès aux attributs d'une classe sont les mêmes que ceux que l'on utilise pour accéder aux champs d'une structure. Les mots clés `public`, `private` et `protected` sont utilisés pour contrôler l'accès aux attributs et aux fonctions des classes; par défaut, le mode d'accès des classes est privé. Donc, ce sont seulement les fonctions membres de cette classe qui peuvent accéder aux attributs privés de la classe. Ceci est le principe important de la programmation objet, alors que les attributs publics sont visibles partout dans le programme.

Pour faire un lien entre les services offerts par la classe et l'utilisateur d'un logiciel, on a défini des méthodes publiques. L'ensemble de ces services constitue l'interface publique

de cette classe. Alors, les utilisateurs d'une classe n'ont plus besoin de se préoccuper de la manière dont s'exécutent les tâches. D'ailleurs, ils ne peuvent accéder ni aux membres privés de la classe, ni à ses fonctions publiques. Ces dernières font partie de l'implémentation de cette classe.

L'accès à des données privées doit être rigoureusement contrôlé par les fonctions membres, aussi appelées fonctions d'accès. Par exemple, pour permettre la lecture d'une donnée privée, la classe peut utiliser une fonction `get`, alors que, pour modifier ces données, la classe peut utiliser la fonction *set*. De telles modifications semblent violer la notion de données privées.

Mais la fonction membre *set* est capable de faire les validations nécessaires pour s'assurer que la donnée est bel et bien modifiée Quant à la fonction `get`, son rôle est d'éditer et de limiter la visibilité d'une donnée, de manière qu'elle soit visible par tous les clients qui auront à l'utiliser. Les fonctions n'ont pas besoin d'être déclarées publiques pour faire partie de l'interface d'une classe : une fonction peut être déclarée privée et être utilisée comme utilitaire par d'autres fonctions de la classe.

6. Conception par objets : étude de cas

Contrairement à la programmation fonctionnelle classique, la programmation par objets est centrée sur les structures de données manipulées par le programme. Voyons un exemple de programmation par objets, afin d'en clarifier le principe.

Un logiciel doit gérer un dessin graphique interactif dans un système de multifenêtrage. Ce dessin est composé d'objets graphiques qui, une fois sélectionnés à l'aide de la souris, réagissent chacun d'une manière bien spécifique. Ainsi, le dessin doit pouvoir gérer l'apparition et la disparition d'éléments graphiques, être imprimé et transmettre les sélections de l'utilisateur aux objets désignés. Par exemple, lorsqu'un utilisateur clique dans la fenêtre, le dessin reçoit un message qui lui fournit les coordonnées du point sélectionné. La figure 11 illustre tous ces concepts.

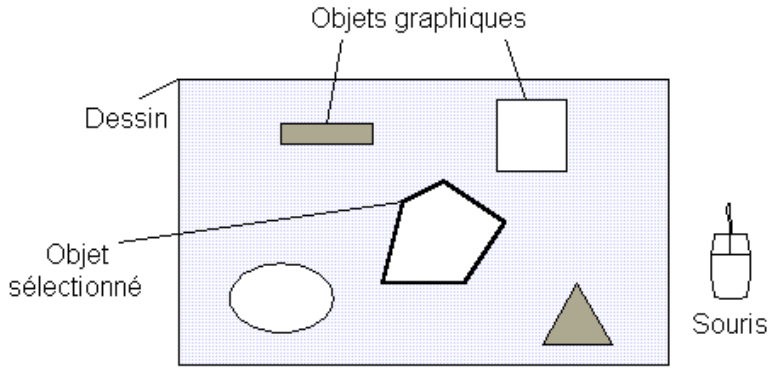

Figure 11 Dessin et objets graphiques à concevoir.

Deux classes d'objets sont alors définies : le dessin et l'objet graphique. La communication entre ces objets se fait par des envois de messages de la forme :

```
objet message(paramètres)
```

Dans le cas de la figure 11, les opérations que l'on peut effectuer sur la classe `Dessin` (ses méthodes) sont :

- `tenir_compte_sélection` (position : coordonnées);
- `imprimer()`.

Quant à la classe `Objets_graphiques`, elle a les méthodes suivantes :

- `tenir_compte_sélection()`, où l'objet doit réagir spécifiquement à sa sélection;
- `contenir_position(position : coordonnées)`, qui retourne une valeur booléenne pour indiquer si l'objet possède la position fournie en paramètre.

Lorsqu'un utilisateur clique dans le dessin, ce dernier est informé de l'envoi d'un message `tenir_compte_sélection()` et exécute le traitement suivant :

```
POUR chaque objet_courant de la classe Objets_graphiques
  SI objet_courant contenir_position(position)
      objet_courant tenir_compte_sélection()
  FINSI
FINPOUR
```

Poursuivons notre exemple en faisant correspondre des noms aux objets graphiques. Nous avons alors :

- les *navettes*, qui sont des objets graphiques mobiles, créés périodiquement par le dessin et qui le traversent de part et d'autre en évitant les objets graphiques situés sur leur trajectoire;
- les *missiles*, qui sont aussi des objets mobiles créés aléatoirement et qui se déplacent vers les navettes les plus proches pour entrer en collision avec elles;
- les *obstacles*, qui sont des objets graphiques immobiles situés dans le dessin et créés au début de l'application; lorsqu'un objet mobile entre en collision avec eux ou lorsqu'ils sont sélectionnés, ils changent de forme;
- les *objets mobiles*, qui se déplacent et qui changent de trajectoire lorsqu'ils sont sélectionnés.

Ces nouvelles classes sont illustrées à la figure 12 et héritent de la classe `Objets_graphiques` qu'elles précisent par :

- leur forme spécifique;
- la redéfinition de certaines méthodes comme `tenir_compte_sélection();`
- les propriétés supplémentaires que possèdent certaines classes, telles que `déplacer()`.

Le principe de fonctionnement du système est simple : la classe `Dessin` aura une méthode appelée `traiter_cycliquement` qui sera activée cycliquement et agira comme un séquenceur par rapport aux objets mobiles. Une telle méthode exécute le traitement suivant :

```
SI un intervalle de temps est écoulé
      navette create()
      missile create()
FINSI
Pour chaque objet_courant parmi les Objets_graphiques
      SI objet_courant appartient à la classe Objets_mobiles
            objet_courant déplacer()
      FINSI
FINPOUR
```

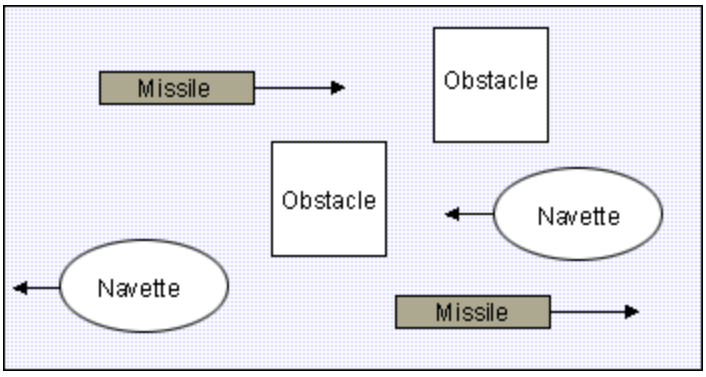

Figure 12 Navette, missile et obstacle.

Par ailleurs, les objets mobiles doivent tenir compte des autres objets graphiques pour réagir. La navette doit éviter les obstacles, alors que le missile vise les navettes. Cela amène à la définition des méthodes suivantes :

- `Présence(trajectoire_souhaitée : trajectoire)`, qui retourne une valeur booléenne pour indiquer si l'objet courant est sur la trajectoire fournie;
- `Déterminer_position()`, qui spécifie la position de l'objet courant.

L'ensemble des classes et de leurs méthodes donne alors lieu au modèle de la figure 13.

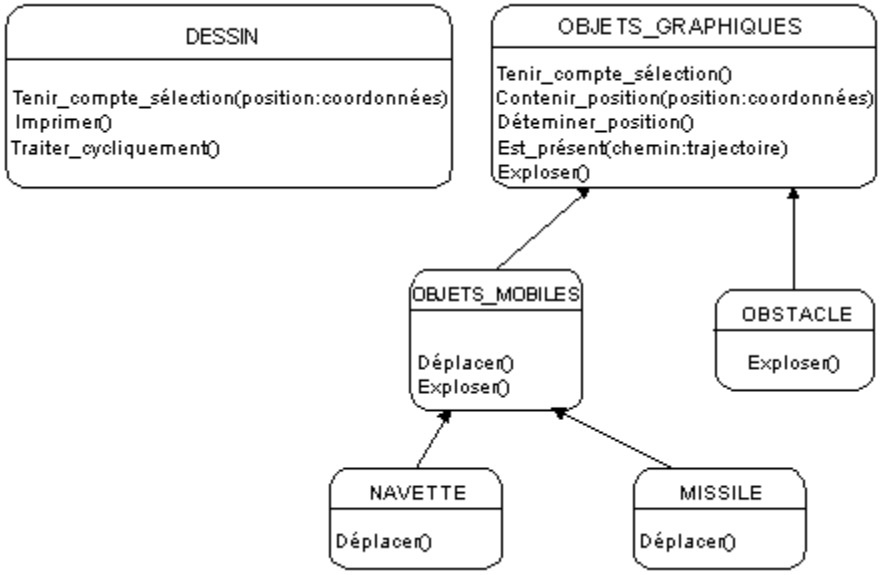

Figure 13 Classes de l'application graphique.

Cet exemple montre comment la conception par objets mène à une structure simple organisée autour des classes. Ces dernières n'ont pas besoin d'être complètes au moment de la conception du programme, mais peuvent être élaborées progressivement. La simplicité de l'implémentation suggère des idées intéressantes pour la spécification des modules.