1 Introduction

2 Complexité des problèmes

2.1 Évolution de la mécanique d'abstraction par les langages
2.1.1 Programmation non structurée
2.1.2 Programmation procédurale
2.1.3 Programmation modulaire
2.1.4 Programmation orientée objets
2.1.5 Généalogie des langages: exemples de langage et évolution

2.2 Gestion de la complexité : 3 principes fondamentaux
2.2.1 Abstraction : l'essentiel d'un objet
2.2.2 Hiérarchie : organisation
2.2.3 Décomposition : diviser pour régner

3 Modèle objet

3.1 Buts du modèle objet

3.2 Changement de mentalité

3.3 Composantes de l'objet

3.4 Qu'est-ce qu'une classe?

3.5 Encapsulation

3.6 Interface
3.6.1 Catégories d'opérations sur un objet

3.7 Relations entre classes

4 Conclusion

1 Introduction

Le but de ce cours d'introduction est de vous présenter les notions essentielles de la programmation orientée objets. Pour la plupart des gens, la programmation en C ou en Fortran n'utilise pas les concepts orientés objets. L'étape la plus difficile pour un nouveau programmeur en orientée objets est de changer sa « philosophie de programmation ». Le passage à l'orientée objets force en effet le programmeur à aborder les problèmes sous un nouvel angle, très différent de la programmation « séquentielle classique ».

Toutefois, cette nouvelle approche tend à rapprocher la programmation des concepts ou objets réels. C'est donc avec le goût de changer votre façon de programmer et de voir les choses que vous devez lire ces notes de cours. Si vous y arrivez, le monde orienté objets vous appartiendra… Si vous n'y arrivez pas, vous aurez de la difficulté à programmer et à comprendre les programmes déjà faits.

2 Complexité des problèmes

Le dernier ordinateur d'IBM (septembre 2000) peut faire un trillion (10^{18}) d'opérations à la seconde. Si l'on demandait à quelqu'un de faire de même avec une calculatrice, il lui faudrait 10 années pour faire l'équivalent d'une seule seconde pour cet ordinateur.

Les problèmes que nous désirons résoudre deviennent de plus en plus complexes. Au début, le simple résultat d'un calcul était intéressant. Maintenant, le résultat brut d'un calcul ne nous intéresse plus! Il faut qu'il soit affiché graphiquement et automatiquement utilisé pour résoudre le problème d'ordre plus global dont il fait partie.

Les 1ers programmes pouvaient être réalisés par une seule personne et faits en assembleur. Aujourd'hui, les gros logiciels peuvent être réalisés par des dizaines, voire des centaines de personnes! Sans compter que les outils utilisés par ces personnes sont eux-mêmes développés par autant de programmeurs.

L'apparition de langages de plus haut niveau permet aujourd'hui de traiter des problèmes beaucoup plus complexes. Avec l'apparition de ces problèmes plus complexes, il devient de plus en plus difficile, voire impossible pour un seul individu de comprendre tous les aspects du problème à traiter.

"The world is only sparsely populated with geniuses. There is no reason to believe that the software engineering community has an inordinately large proportion of them." -- Peters, L. 1981. Software Design.

Puisque nous ne pouvons nous fier sur les génies, il faut développer des outils et des méthodes de travail qui nous permettent d'atteindre nos objectifs et de résoudre ces problèmes plus complexes.

La programmation orientée objets est un outil performant qui nous permet d'affronter de façon efficace les tâches de programmation les plus ardues.

2.1 Évolution de la mécanique d'abstraction par les langages

2.1.1 Programmation non structurée

La programmation non structurée est le 1er type de programmation à être apparu. Elle présente certaines caractéristiques qui lui sont propres :

- données globales;
- grand risque associé au partage des données globales;

- difficile à maintenir la cohérence dans le temps.

2.1.1.1 Données globales

Les données sont connues et utilisables par toutes les parties du code. Elles sont donc très « vulnérables » dans n'importe quelle partie du programme. Tout le monde peut modifier ces données sans qu'il n'y ait de vérifications de faites. On n'est donc jamais sûr de leur « validité ».

2.1.1.2 Grand risque associé au partage des données globales

Par exemple, avec une matrice globale, n'importe qui peut décider de modifier le contenu de la matrice, dans n'importe quelle fonction. Ceci pourrait être extrêmement nuisible pour les utilisateurs de cette matrice, qui pourraient s'attendre à ce que son contenu demeure inchangé durant certaines opérations comme l'écriture sur disque.

2.1.1.3 Difficile à maintenir la cohérence dans le temps

Puisqu'il est difficile de savoir qui modifie les données et que les opérations sont codées plusieurs fois dans le programme, il est difficile de maintenir le programme et d'y apporter des modifications. Une fois qu'il est terminé et qu'il fonctionne, il évolue difficilement.

2.1.2 Programmation procédurale

La programmation procédurale amène quelques nouveautés à la programmation non structurée. Elle permet notamment de mieux ordonner les programmes. On voit apparaître les procédures et les fonctions.

- Apparition des procédures et fonctions
- Réutilisation de fonctions dans plusieurs contextes
- Division de la tâche en plusieurs sous-tâches

2.1.2.1 Apparition des procédures et fonctions

On rassemble dans des « sous-programmes » séparés les calculs ou opérations que l'on exécute souvent. On nomme ces « sous-programmes » des procédures ou des fonctions. Si par exemple on a une opération qui prend une vingtaine de lignes, on rassemble ces lignes dans la fonction et on donne un nom unique à cette fonction. On remplacera alors dans le reste du programme les 20 lignes que prenait l'opération par un simple appel à la fonction créée.

Ceci a aussi pour effet que lorsqu'un bogue est découvert dans une fonction, on le corrige à un seul endroit plutôt qu'à plusieurs.

2.1.2.2 Réutilisation de fonctions dans plusieurs contextes

On utilise l'avantage que la fonction peut travailler avec les paramètres qui lui sont passés, au lieu de travailler sur des données globales. Ceci a pour effet de permettre la réutilisation de la fonction dans d'autres contextes. Il se peut même que certains programmeurs utilisent la fonction dans un contexte que le concepteur n'aurait même pas prévu!

2.1.2.3 Division de la tâche en plusieurs sous-tâches

Dans l'esprit de regroupement des opérations sous des fonctions, le programme complet sera transformé par des appels de fonctions. De plus, dans une fonction, on pourra appeler une autre fonction. Il en résultera l'apparition de fonctions de plus « haut niveau », c'est-à-dire des fonctions qui effectueront des tâches de plus en plus complexes, en faisant elles-mêmes appel à des fonctions plus simples.

2.1.3 Programmation modulaire

La programmation modulaire nous permet d'utiliser des morceaux de programmes ou des ensembles de fonctions (librairies) déjà compilés, voire vendus par une tierce personne. Il est aussi plus facile de comprendre chaque module séparément, donc un à un, plutôt que de comprendre l'ensemble du programme.

- Compilation séparée
- Compréhension plus facile
- Plus appropriée pour les équipes de développement

2.1.3.1 Compilation séparée

L'un des avantages des librairies, c'est que la compilation est séparée, c'est-à-dire que l'utilisateur n'a pas à compiler le code source des fonctions, puisque cela a déjà été fait, soit par le vendeur ou par quelqu'un d'autre. Souvent, il arrive que l'utilisateur de la librairie n'ait pas le code source des fonctions qui la composent. Il doit dans ce cas faire « confiance » au vendeur. Le vendeur en contrepartie doit fournir une documentation de chacune des fonctions de la librairie et en expliquer l'utilisation.

2.1.3.2 Compréhension plus facile

En rassemblant les fonctions dans des librairies, on choisira un « thème » commun à chacune de ces fonctions. Ainsi, on pourra rassembler toutes les fonctions mathématiques dans une librairie portant le nom `libm.a` (ce qui est le cas sous Unix). De plus, lorsqu'un programmeur a besoin de faire une opération reliée aux mathématiques, il saura qu'il lui faut aller chercher dans la librairie mathématique. S'il a besoin de fonctions graphique 3D, il pourra aller voir dans la librairie graphique (Ex : `libGL.a` pour OpenGL).

2.1.3.3 Plus appropriée pour les équipes de développement

Dans le cadre d'un projet de développement de logiciel, chaque personne travaille souvent sur une partie précise du projet et plutôt rarement sur l'ensemble. On peut donc regrouper les gens qui travaillent aux même parties du logiciel et faire des modules avec chacune de ces parties de logiciel. Ainsi, le groupe de développeurs de la librairie mathématique fournira une librairie mathématique pour tous les autres développeurs du projet. Cette librairie mathématique pourra être réutilisée par le groupe de développeurs de la partie graphique qui va avoir besoin de certaines fonctions.

On peut donc rendre plus indépendant le développement des différentes composantes d'un logiciel.

2.1.4 Programmation orientée objets

Le passage de l'orienté procédure à l'orienté objets amène plusieurs aspects intéressants :

- peu ou pas de variables globales;
- nouvelle façon de penser la décomposition de problèmes;
- les programmes deviennent un ensemble d'objets faiblement couplés.

2.1.4.1 Peu ou pas de variables globales

Dans la programmation orientée objets, on n'aura pas besoin (du moins en théorie) de variables globales. Ceci est principalement dû au fait qu'en orienté objets, le travail est fait dans un environnement contrôlé qu'est la classe. Chaque fonction qui existait dans le monde procédural a été associée à une classe et c'est dans cette classe que se retrouveront aussi les données avec lesquelles ces fonctions vont travailler. Les variables globales deviendront soit des attributs de la classe, ou des paramètres qui seront passés à la fonction de la classe (que nous appellerons méthode).

2.1.4.2 Nouvelle façon de penser la décomposition de problèmes

L'orienté objets n'est pas seulement l'ajout de mots dans un langage, c'est un nouvelle façon de penser la décomposition de problèmes et l'élaboration de solutions. En effet, la programmation orientée objets nous force à voir le problème sous un nouvel angle, à faire de nouvelles associations entre les fonctionnalités et ceux qui les possèdent. Dans le monde orienté objets, chaque objet a un état, une identité propre ainsi qu'un comportement. Nous reviendrons sur ce point plus tard.

En programmation procédurale, on pensait en termes de fonctions qui devaient effectuer un certain travail dans un ordre précis. En orienté objets, on pense plutôt à de petits objets distincts et détachés qui peuvent chacun exécuter un travail restreint à leur domaine de compétence. Un exemple simple serait une classe de matrice. C'est la classe matrice qui sait comment s'additionner à une autre matrice, et non pas une fonction externe. Tout cela semble plutôt philosophique et ça l'est aussi!

2.1.4.3 Les programmes deviennent un ensemble d'objets faiblement couplés

Lorsque l'on pense un programme en termes d'objets, on restreint les fonctionnalités dans ces objets. Chaque objet saura accomplir une série de tâches qui lui est propre. Les programmes feront alors affaire avec ces objets pour les fonctionnalités qu'ils offrent.

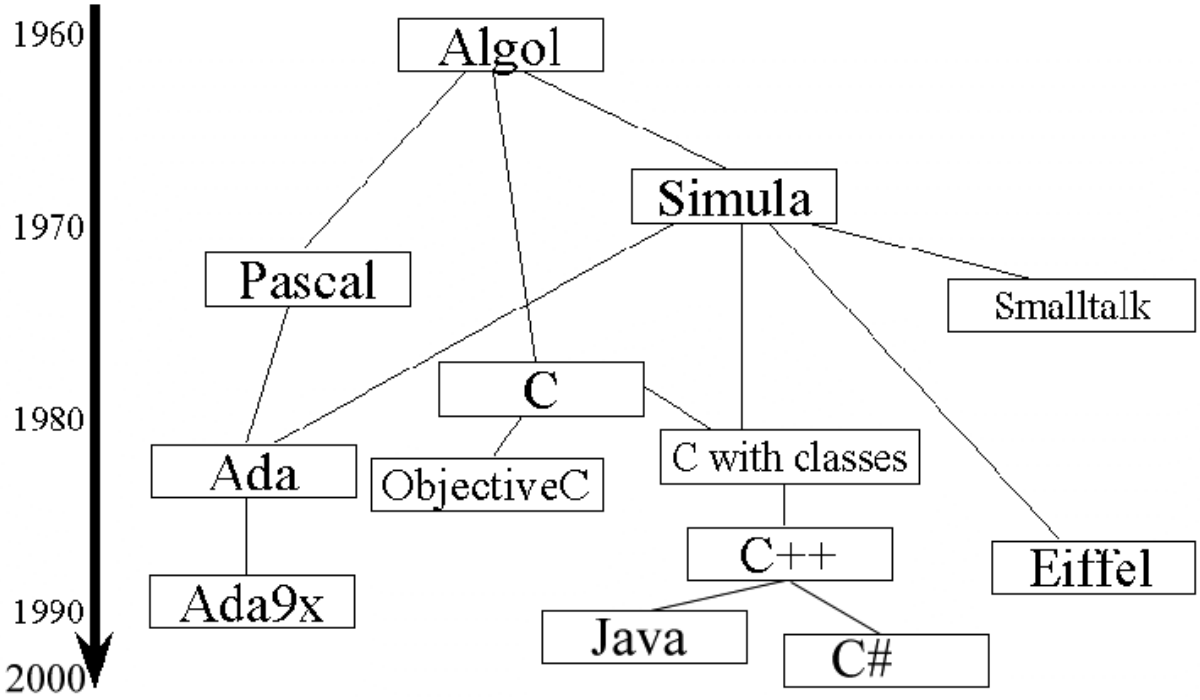
Puisque chaque objet offre ses propres fonctionnalités, il peut alors exister sans ne rien connaître des autres objets dans le programme. On dit alors qu'il est indépendant, ou non couplé aux autres objets. Toutefois, l'ensemble du programme représente justement l'interaction entre les différents objets utilisés.

Certains objets sont eux-mêmes composés d'autres objets ou font appel à d'autres objets pour remplir certaines tâches. Ils sont alors plus ou moins faiblement couplés.

Autre exemple, au lieu d'avoir un programme totalisant 2000 fonctions, on aura plutôt 100 classes avec 20 fonctions chacune.

2.1.5 Généalogie des langages: exemples de langage et évolution

Graphique de l'évolution des langages



2.2 Gestion de la complexité : 3 principes fondamentaux

L'approche orientée objets permet de créer l'illusion de simplicité. Il faut rendre le code de programmation simple à utiliser pour l'usager. Cela veut souvent dire rendre le code tant du point de vu nomenclature qu'utilisation le plus près possible de la réalité. Pour rendre le tout simple, le programmeur orienté objets devra faire un travail parfois non négligeable, car la programmation de certains concepts peut parfois demander un effort appréciable.

Prenons l'exemple du téléphone. Lorsque vous téléphonez à quelqu'un, vous utilisez l'objet téléphone, que vous décrochez, ensuite vous composez le numéro et puis vous parlez si l'autre personne répond. En tant qu'utilisateur, cela vous importe peu de savoir que lorsque vous décrochez le combiné vous ouvrez un circuit à la centrale, que votre signal à tonalité est interprété par un ordinateur, que les signaux de trop haute fréquence sont filtrés, qu'il y a une vérification dans la centrale sur le numéro que vous composez si c'est un appel interurbain ou non et que votre voix est convertie d'un signal analogique à numérique pour pouvoir voyager par fibre optique.

Les compagnies de téléphone ont comme but de vous offrir un service simple à utiliser. Il en va de même pour les programmeurs orientés objets : leur but est d'offrir du code de programmation simple à utiliser.

Pour aider à offrir du code simple à utiliser, la programmation orientée objets se base sur 3 principes :

- l'abstraction;
- la hiérarchie;
- la décomposition.

2.2.1 Abstraction : l'essentiel d'un objet

Une abstraction définit les caractéristiques essentielles d'un objet qui le différencie de tous les autres objets. Ceci est toujours relatif à la perspective de l'observateur.

Le rôle de l'abstraction joue pour beaucoup dans la construction d'un programme simple. Dans l'exemple de l'appel téléphonique, on veut faire abstraction de toute l'électronique qu'il y a derrière l'utilisation d'un téléphone. On va donc ignorer les détails non essentiels, pour ne retenir que ceux qui nous apparaissent comme essentiels.

Ceci vient du fait que l'humain a une limitation naturelle pour gérer la complexité. On peut manipuler entre 5 et 9 concepts à la fois. Plus on approche de cette limite, plus il nous faut du temps pour saisir, mémoriser.

On ignore les détails non essentiels :

on parle de téléphone et non de circuits électroniques;
on parle de forêt et non plus d'arbres;
de plage et non de grains de sables;
de maison et non de briques.

Donc pour les gens en général, les caractéristiques d'un téléphone sont :

- un combiné composé d'un micro et d'un haut-parleur;
- un clavier numérique.

2.2.2 Hiérarchie : organisation

La hiérarchie est l'organisation des connaissances sous forme pyramidale. C'est diviser les comportements d'une classe de façon hiérarchique. Cela permet une compréhension plus rapide du problème. C'est aussi un système de classification.

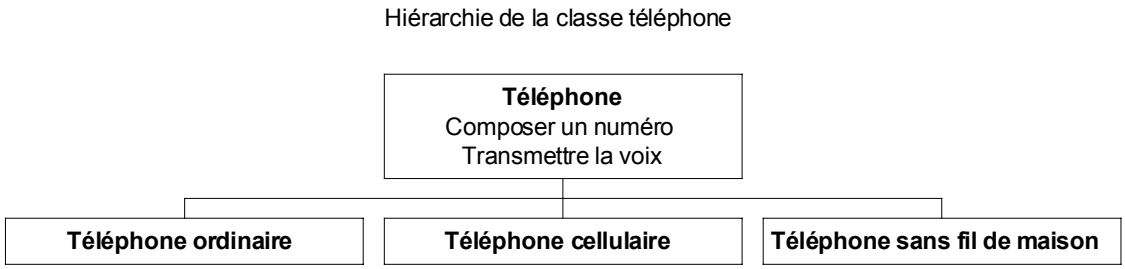
Par exemple, lorsque je vous parle de la notion de téléphone, vous pensez probablement à l'un des téléphones que vous avez chez vous. Bien que ce soit tous des téléphones différents, ils permettent tous au moins d'appeler quelqu'un.

Maintenant, il existe plusieurs sortes de téléphone :

- cellulaires;
- sans fil de maison;
- ordinaires.

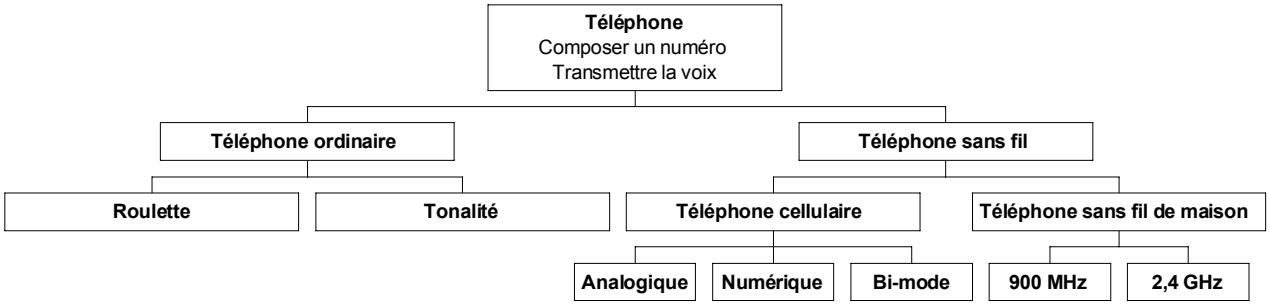
Tous ces téléphones peuvent être regroupés sous l'idée de « téléphone ».

On pourrait les regrouper comme suit :



On peut cependant vouloir diviser encore plus les idées, en regroupant les différents téléphones en sous-catégories.

Hiérarchie de la classe téléphone plus détaillée



2.2.3 Décomposition : diviser pour régner

Pour définir et développer un système, il est essentiel de décomposer le sujet en petits morceaux. Nous pourrons ensuite les définir séparément.

Reprenons l'exemple du téléphone. Prenons un téléphone ordinaire à tonalité. On pourrait décomposer cet objet en plusieurs sous-objets qui seront eux-mêmes redécomposés en plusieurs autres sous-objets. On pourrait voir le téléphone comme une composition d'un clavier, d'un micro, d'un haut-parleur, d'un fil. Si on prend chacun de ces sous-objets, on pourrait les redécomposer. Par exemple, le clavier est composé de 12 touches. Chaque touche est composée d'un bout de matériau (plastique ou caoutchouc) ainsi que d'un symbole numérique ou du # et *. On pourrait aller encore plus loin en disant que le plastique est composé de molécules qui sont elles-mêmes faites d'atomes.

Lorsque nous voulons comprendre une application qui est composée de plusieurs objets différents, nous avons à comprendre seulement quelques parties, pas tout le système à la fois.

3 Modèle objet

3.1 Buts du modèle objet

Le but du modèle objet :

- s'éloigner de l'espace de la machine et se rapprocher de l'espace du problème;
- développer une approche mieux adaptée à la résolution de la plupart des problèmes;
- on passe de l'orienté action à l'orienté objets.

L'approche orientée objets nous permet de nous rapprocher de l'espace du problème, c'est-à-dire que lorsqu'on programme le problème que l'on veut résoudre, on utilise une syntaxe et une terminologie très près de celle que l'on utilise lorsqu'on écrit la démarche sur papier. Par exemple, si on veut calculer l'aire d'un triangle, on va écrire sur papier : « calcul de l'aire du triangle ». En orienté objets, on dirait que l'on demande au triangle de nous calculer son aire (Ex : `lAire = lTriangle.calculAire()`).

Si on veut calculer l'aire totale d'une surface composée de triangles, on devrait alors boucler sur tous les triangles, demander à chacun de calculer son aire et faire la somme de toutes ces aires.

La modélisation objet nous permet aussi une lecture plus facile du programme puisque celui-ci est écrit de la même façon que le problème est défini.

En développant ainsi des objets qui répondent aux questions que nous désirons leur poser, on développe un modèle ou programme mieux adapté à la résolution de la plupart des problèmes. On passe d'une programmation orientée actions à une programmation orientée objets. Exemple : supposons que l'on vous demande d'écrire un programme qui lit un fichier de triangles et qui calcule la surface de chacun de ces triangles et l'affiche à l'écran. Dans un programme orienté actions, on aurait :

```
for i = 1:nb_triangles
      XYZ = fscanf(fid, `%g`,9);
      X = XYZ([1 4 7]);
      Y = XYZ([2 5 8]);
      Z = XYZ([3 6 9]);
      lAire = surf_tri(X,Y,Z);
      fprintf(`Aire : %g`, lAire);
end
```

Dans un programme orienté objets, on aurait :

```
Triangle lTri;
for i = 1:nb_triangles
      lTri.lecture(fid);
      lAire = lTri.calculSurface();
      lEcran <<`Àire :` << lAire;
end
```

3.2 Changement de mentalité

La programmation orientée objets demande de faire un changement de mentalité. On se concentre sur l'objet et non plus sur la fonction. On se pose la question : qu'est-ce que cet objet est capable de faire? La programmation devient plus naturelle. Vous savez que si vous avez un objet « téléphone », vous pouvez lui demander de composer un numéro. C'est normal. Si vous avez un objet « triangle », vous ne lui demanderez pas de composer un numéro de téléphone... On demande aux objets qui ont un certain savoir-faire d'exécuter le travail qu'ils savent faire.

Dans les langages procéduraux comme Matlab, Fortran, C, on programme des fonctions. L'unité de programmation est la fonction. On a des données, on se demande quoi faire avec et on trouve une fonction qui va traiter ces données et nous retourner les résultats attendus. A priori, on pourrait appeler n'importe quelle fonction sur n'importe quelle donnée.

Dans la programmation orientée objets, l'unité de programmation est la classe. La classe réunit les données et les fonctions que l'on peut utiliser avec ces données. Une classe peut donner naissance à plusieurs objets. C'est comme la classe triangle : on sait de quoi est composé un triangle et ce qu'il peut faire, par contre, une surface peut être composée de milliers de triangles différents. Lorsqu'on prend un triangle en particulier, on parle d'un objet ou d'une instance de la classe triangle. Tout le monde ici peut avoir un téléphone ordinaire à tonalité de la même marque, de la même couleur, etc. mais chacun de ces téléphones reste unique. Le téléphone rouge de M. Y n'est pas le même que le téléphone rouge de Mme Y. C'est la même sorte, mais pas le même objet. On dit alors que c'est la même classe, mais pas le même objet.

Les programmes orientés objets sont faits d'objets qui se parlent entre eux. Ils s'envoient des messages qui produisent les actions désirées.

3.3 Composantes de l'objet

L'objet se décompose en 3 parties :

- état;
- comportement;
- identité.

L'état d'un objet se traduit par la valeur des variables qui le composent. Comme le triangle est composé de 3 coordonnées, l'état précis d'un triangle pourrait être d'avoir les coordonnées (0,0,0) (1,0,0) et (0,1,0).

Le comportement d'un objet c'est les actions que cet objet peut faire. Par exemple, une action que le triangle peut faire c'est de calculer son aire.

Finalement, l'identité d'un triangle c'est le fait qu'il soit unique parmi tous les autres triangles. Même si 2 triangles ont exactement les mêmes coordonnées, ils demeurent 2 triangles différents, car il en existe 2 en mémoire. De plus, il se peut qu'à un moment donné, l'un de ces triangles change de position, pour se distinguer géographiquement de son jumeau.

3.4 Qu'est-ce qu'une classe?

Lorsqu'on commence à programmer en orienté objets, on a tendance à confondre le concept de classe et d'objet. Qu'est-ce qui distingue ces 2 concepts? Tout d'abord, il faut savoir qu'un objet est une entité discrète qui existe dans le temps et l'espace. On veut par exemple désigner un triangle bien précis, avec des coordonnées connues. Le concept de l'objet désigne donc UN objet en particulier.

Le concept de classe quant à lui n'est qu'une abstraction qui doit représenter un ensemble d'objets qui partagent une structure commune et un comportement commun. Par exemple, lorsqu'on discute des triangles en général, sans faire mention d'un triangle bien précis, on peut dire que nous discutons de la classe triangle. L'objet est une sorte d'incarnation d'une classe, c'est-à-dire le concept général qui prend forme en un objet bien précis, avec son état particulier.

La classe définit les opérations qui sont permises sur les objets de la classe, c'est-à-dire que, lorsque l'on parle des téléphones en général (de la classe), on peut dire qu'ils peuvent tous servir pour transmettre la voix et pour composer un numéro. Lorsqu'on utilise un téléphone en particulier (un objet), on peut à ce moment-là vraiment faire l'action de composer le numéro de téléphone et transmettre la voix. Ce n'est qu'avec un objet concret que l'on peut faire des actions concrètes. La classe n'est là que pour définir ce qu'il est possible de faire.

Elle définit aussi les états que les objets peuvent prendre. Par exemple, nous savons que tous les triangles ont 3 coordonnées X, Y et Z. Si la classe triangle est définie ainsi, aucun objet triangle issu de cette classe ne pourra avoir des coordonnées en X, Y, Z et T. Tous les objets triangles issus de la classe-mère triangle auront nécessairement 3 coordonnées X , Y et Z. De plus, la classe peut contraindre les valeurs que peuvent prendre ces coordonnées. Par exemple, on pourrait ajouter une restriction dans la classe triangle pour qu'aucune coordonnée ne s'y superpose. Ainsi, on pourrait contrôler la validité des données d'un objet à toutes les fois qu'on l'utilise.

La classe définit donc principalement 2 choses :
- les données qu'elle peut contenir (attributs);
- les fonctions que l'on peut appliquer sur ces données (méthodes).

Si l'on reprend notre classe triangle, on peut dire qu'elle est définie comme suit :

```
classe Triangle :
```

```
attributs :
aX1,aY1,aZ1
aX2,aY2,aZ2
aX3,aY3,aZ3
```

Méthodes :

```
      calculeBarycentre()
      calculePerimetre()
      caluleNormale()
      calculeSurface()
```

Objets :

```
TriangleDroit :
aX1 = 0, aY1 = 0, aZ1 = 0
aX2 = 1, aY2 = 0, aZ2 = 0
aX3 = 0, aY3 = 1, aZ3 = 0

TriangleQc :
aX1 = 4.5, aY1 = 5.6, aZ1 = 2
aX2 = 2.1, aY2 = 1.1, aZ2 = 1
aX3 = 3.3, aY3 = 0.5, aZ3 = 0
```

3.5 Encapsulation

L'encapsulation est un concept très important dans la programmation orientée objets. C'est avec l'encapsulation des données que l'on va éliminer les variables globales. Qu'est-ce que l'encapsulation? C'est simplement le fait que les données qui forment un concept sont réunies dans un objet, aussi complexes soient-elles. Il faut donc faire un effort de réflexion au moment de la conception de la classe pour penser à réunir les bonnes données ainsi que les fonctions qui s'y rattachent. Comme nous l'avons dit plus haut, les données réunies dans une classe s'appellent les attributs et les fonctions d'une classe sont les méthodes.

Dans les langages procéduraux, il n'y a aucun couplage entre les données et encore moins entre les données et les fonctions. Dans les langages orientés objets, on obtient un fort couplage entre les données ainsi qu'avec les fonctions qui s'y appliquent. Ceci nous permet entre autres, de « masquer » la complexité des opérations qui s'effectuent dans les méthodes. En rendant ainsi les objets plus « simples », l'utilisateur pourra développer des applications de plus en plus puissantes tout en diminuant le temps de développement.

Un très grand avantage de l'encapsulation est la possibilité d'évolution d'une classe. Puisque les données sont « cachées » dans celle-ci, il nous est possible de modifier leur nature à mesure que les besoins évoluent. Dans un langage orienté objets, cela aura peu d'impact. Par contre, dans un langage procédural, si l'on désire changer la structure des données, cela sera désastreux pour tout le programme. Par exemple, si nous avons une classe Triangle qui a 9 attributs réels. Si l'on désire changer cela pour une matrice 3x3, on ne fait que les changements dans la classe elle-même ainsi que dans les méthodes de la classe. Tous ceux qui utilisent les méthodes de la classe ne seront pas affectés par ce changement.

Ex :

```
classe Triangle :

attributs : // Partie cachée
      float aXYZ[3,3];

Méthodes : // Partie « publique » qui ne change pas!
      calculeBarycentre();
```

```
      calculePerimetre();
      caluleNormale();
      calculeSurface();
```

3.6 Interface

L'interface d'une classe définit ce qui est possible de faire avec les objets de cette classe. C'est la liste des opérations possibles sur les objets. Dans l'interface, on ne retrouve que ce qui est nécessaire à l'usager, donc on cache les détails inutiles ou complexes.

```
class Triangle
{

public: // Interface publique
      void    asgnCoordonnees  (X[],Y[],Z[])
      float[] calculeBarycentre();
      float   calculePerimetre ();
      float[] caluleNormale    ();
      float   calculeSurface   ();

private: // Partie cachée
      float aXYZ[3,3];
```

L'interface se construit au fur et à mesure que le programmeur développe sa classe. Au début on commence par y inclure un minimum de fonctionnalités et on en ajoute de nouvelles au besoin.

3.6.1 Catégories d'opérations sur un objet

Par ses méthodes, un objet offre différents services que l'on peut catégoriser.

- **Constructeur :** sert à initialiser l'objet à une valeur initiale, par défaut.
- **Destructeur :** sert à libérer les ressources du système.

Les constructeurs et destructeurs sont des opérations très importantes. Le constructeur sert à initialiser correctement les données portées par l'objet. La validité des données est ainsi assurée dès le tout début de l'existence de l'objet.

Le destructeur quant à lui sert à libérer les ressources du système. Par exemple, pour un objet de type matrice, celui-ci doit dans son destructeur libérer l'espace mémoire de la matrice elle-même.

- **Accès :** sert à consulter de l'information (attributs) dans l'objet sans le modifier.
- **Assignation :** sert à modifier l'information (attributs) dans l'objet.

Les opérateurs d'accès nous permettent typiquement de consulter les attributs d'un objet. Par exemple, si on avait l'objet « Triangle » et que l'on voulait connaître les coordonnées de son 1er point, on pourrait définir une méthode de la classe triangle qui nous retournerait un vecteur contenant [X1, Y1, Z1].

```
[pX1, pY1, pZ1] Triangle::reqPoint1() {
        return [pX1, pY1, pZ1]
}
```

Les opérateurs d'assignation servent à modifier les attributs d'un objet par des valeurs passées en argument. Par exemple, pour affecter le 1er point d'un triangle, on pourrait avoir une méthode de la classe « Triangle » qui affecterait les valeurs des attributs aX1, aY1 et aZ1, par la valeur des paramètres passés :

```
void Triangle::asgnPoint1(pX1, pY1, pZ1) {
        aX1 = pX1;
```

```
        aY1 = pY1;
        aZ1 = pZ1;
}
```

- **Comparaison :** pour comparer un autre objet du même type avec lui-même
- **Itérateur :** pour parcourir des ensembles (conteneurs).
- **Copie & clonage :** copie à partir d'un autre objet ou copie de l'objet lui-même.
- **Entrée/Sortie :** pour écrire ou lire l'objet.

Les opérations d'entrée/sortie sont nécessaires pour pouvoir lire l'objet à partir d'un objet d'entrée (fichier, clavier, réseau), ou écrire l'objet dans un objet de sortie (fichier, écran, imprimante).

3.7 Relations entre classes

Association : lien entre 2 classes (le plus faible)
Agrégation : un processeur fait partie d'un ordinateur
Héritage : une chat est un mammifère

4 Conclusion

La programmation orientée objets nous oblige à faire un changement de mentalité profitable.

Elle a été largement adoptée par l'industrie, ce qui en fait un choix incontournable.

Pour apprendre le C++, vous pouvez consulter le site suivant :

http://www.giref.ulaval.ca/~ericc/cpp