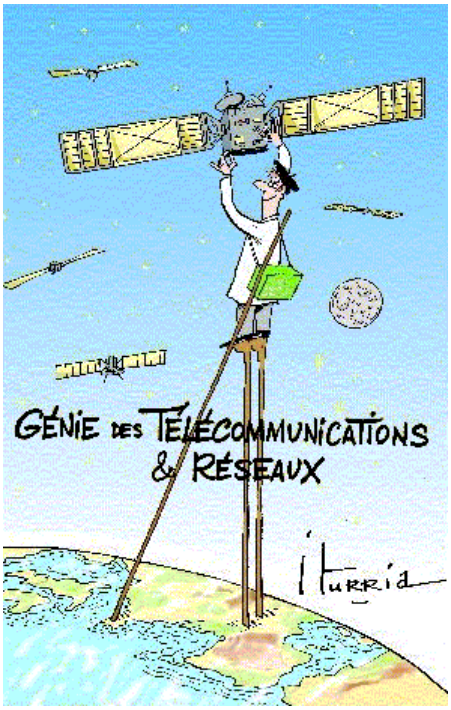


Université de Pau et des Pays de l'Adour

Institut Universitaire de Technologie des Pays de l'Adour

Département Réseaux et Télécommunications

371, rue du Ruisseau
BP 201
40004 Mont-de-Marsan Cedex
tél : 05 58 51 37 00
fax : 05 58 51 37 37

Programmation Orientée Objet

Manuel Munier
E-mail: manuel.munier@univ-pau.fr

Version du 5 septembre 2008

Table des matières

Cette première partie du cours est consacrée à la présentation des concepts communs à l'approche objet. Bien que nous nous focaliserons sur la programmation orientée objet (POO), il faut toutefois savoir que l'approche objet est également particulièrement intéressante dans d'autres domaines tels que les bases de données[1], les méthodes de conception[2] ou les systèmes distribués[3].

1 Historique

La programmation structurée, développée dans les années 60 et au début des années 70, représentée par des langages tels que C et Pascal, a été conçue dans le but d'organiser les programmes complexes en associant les traitements aux structures de données qu'ils manipulent. Ainsi, une application (ou programme) est constitué d'une partie définissant les structures de données (types, variables), d'une partie définissant les opérations (procédures, fonctions) et enfin d'un programme principal qui appelle les opérations. Naturellement, une procédure ou fonction peut également être à son tour un programme avec ses propres types, variables, fonctions,...Cette approche découpe une tâche en sous-programmes et permet ainsi d'analyser les problèmes de manière descendante. Elle met surtout l'accent sur le traitement des programmes, considérant les données comme des individus de deuxième ordre, tout juste bons à servir d'aliment aux procédures.

Seul, en 1967, le langage SIMULA tentait une autre approche en introduisant la notion de classe et d'objet. Confrontés à des problèmes de simulation (comme la modélisation du trafic d'un port ou l'utilisation des services dans une gare), ses concepteurs avaient trouvé tout naturel de caractériser une entité (un individu, un bateau, un port, un guichet, un employé, une chaîne de production,...) sous la forme d'une entité informatique regroupant à la fois une structure de données et l'ensemble des procédures pour la manipuler : un **objet**. Cette démarche, soutenue par des besoins très pragmatiques, n'en constituait pas moins une petite révolution : elle remettait en cause la sacro-sainte distinction entre données et procédures.

La programmation par objets propose donc une vue différente : un programme devient un ensemble de petites entités informatiques qui interagissent et communiquent par messages. Cette conception met en avant l'autonomie de chacune de ces entités informatiques que l'on appelle des objets, en considérant que la complexité croissante des programmes ne peut être réalisée que par des traitements locaux. La gestion de la globalité devient alors un effet secondaire des interactions locales.

Les principes sous-jacents de la programmation par objets sont très simples et peuvent être appréhendés rapidement. En revanche, cette technique requiert de la part du programmeur un changement d'état d'esprit, ceci afin qu'il puisse penser en des termes issus de cette technique.

2 Objectifs

Si la programmation structurée s'est révélée être bien adaptée au développement d'applications complexes et peu évolutives, elle atteint en revanche ses limites lorsque les structures de données ou les fonctions doivent être partagées par différents programmes (à l'aide d'ordres d'import/export) et que les données évoluent. Dans ce cas, une modification d'une structure de données doit être répercutée sur tous les programmes qui la manipulent.

Prenons par exemple le cas du type date représentant une date sous la forme d'une chaîne de six caractères (le 15 janvier 1999 sera codé par la chaîne `"990115"`). Supposons que l'on veuille maintenant coder une date

[1] Les objets apparaissent dans les bases de données soit au travers d'extensions aux systèmes relationnels (Oracle, Informix), soit au travers de nouveaux systèmes généralement construits à partir de langages à objets (O2, ObjectStore).

[2] De nouvelles méthodes de conception basées sur un modèle objet apparaissent (UML). Dans certains cas, il s'agit d'un regroupement de méthodes plus ou moins complémentaires (UML est une unification des notations OMT (J.Rumbaugh) et Booch).

[3] Dans le domaine des systèmes distribués orientés objet, les travaux les plus importants sont ceux de l'OMG autour du *middleware* CORBA. Cette architecture permet de construire des applications distribuées par assemblage de composants objets sur un réseau, voire sur Internet.

sur un triplet d'entiers, i.e. `(15,01,1999)`. Cela va nécessiter de reprendre tous les programmes utilisant ce type pour modifier les bouts de code qui manipulent des dates, tant pour passer d'une suite de caractères à des entiers que pour gérer le passage des années de deux à quatre digits (ex : comparaison de deux dates). Et il ne s'agit pas uniquement d'un cas d'école...

2.1 Modularité

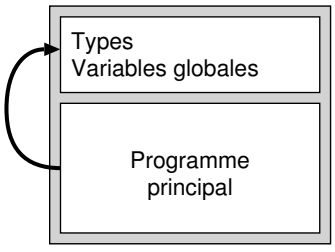
En génie logiciel, on considère que le développement de logiciels de qualité passe par le respect de trois critères essentiels : la **fiabilité**, l'**extensibilité** et la **réutilisabilité** des applications et de leurs composants. Un composant est considéré comme fiable quand on est certain qu'il fonctionne correctement dans tous les cas de figure. L'extensibilité d'une application est la facilité de pouvoir l'étendre (ajout de fonctionnalités) sans avoir à modifier l'existant. Finalement, la réutilisabilité est la capacité des composants logiciels à pouvoir être réutilisés en partie ou en totalité pour construire de nouvelles applications. En fait, nous pouvons résumer tout ceci en un seul mot : **modularité**. Le guide du bon programmeur nous invite donc à respecter les points suivants :

1. **Décomposition/composition** : La décomposition consiste à découper un problème en sous-problèmes (des modules). La composition est la construction, à partir de ces modules élémentaires, de l'application générale. En outre, ces modules pourront éventuellement être utilisés par d'autres applications via le mécanisme d'import/export (cf. `#include` en C).
2. **Compréhension** : La décomposition modulaire doit aider à comprendre le programme, surtout si on n'est pas soi-même le concepteur. L'idée sous-jacente est quand même de réduire la complexité du programme.
3. **Continuité** : Pour que la conception d'un système soit satisfaisante, le nombre de modules devant être modifiés en cas de changement des spécifications (ajout de fonctionnalités, modification des structures de données) doit être restreint.
4. **Protection des données** : Il s'agit d'éviter que quiconque puisse modifier les données d'un module sans y être invité. Les variables globales sont à éviter. On définira plutôt des variables locales aux sous-problèmes. L'idéal serait même de n'accéder aux données d'un module qu'au travers des fonctions qu'il exporte.
5. **Faible couplage** : La réussite de cette technique dépend du degré d'indépendance entre les sous-modules (degré de couplage). On appelle module un sous-système dont le couplage avec les autres est relativement faible par rapport au couplage de ses propres parties.

2.2 Différentes approches

Au fur et à mesure de la complexification des applications à développer, différentes approches sont successivement apparues avec pour objectif d'augmenter le degré de modularité des différents composants (et donc d'améliorer leur fiabilité, leur extensibilité et leur réutilisabilité) :

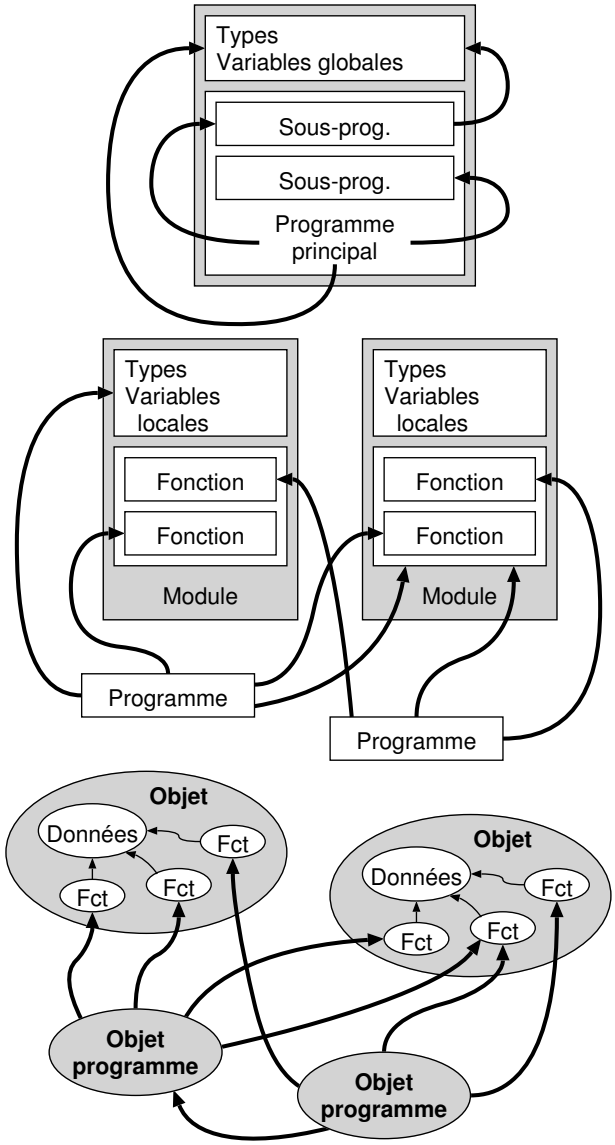
- **Approche brutale** : On définit les types et les variables au début du fichier (dans le meilleur des cas) puis on écrit un unique gros bloc d'instructions dépourvu de tout sous-programme.
  No comment...
  A noter toutefois que cette approche est encore parfois utilisée : assembleur, programmation d'automates, basic,...



– **Approche fonctionnelle** : La modularité ne concerne que les traitements, c'est-à-dire que l'on décompose le programme en sous-programmes (procédures, fonctions) élémentaires. On parle également d'approche descendante.
Au fur et à mesure que le nombre de sous-programmes augmente, il est de plus en plus difficile de s'y retrouver car tout se trouve dans le même fichier.

– **Approche par abstractions** (ou approche structurée) : On découpe l'application en modules, chaque module regroupant les structures de données et les traitements associés (ex : pour le type `pile`, on va regrouper dans un même module la définition du type `pile` et les fonctions `creerPileVide`, `pileVide`, `empiler`, `depiler` permettant de manipuler des piles). Ces modules sont ensuite utilisés via le mécanisme d'import/export (cf. `#include` en C). Toutefois, il est toujours possible d'accéder aux données directement, sans utiliser les fonctions du module.

– **Approche par objets** : On augmente encore l'indépendance entre programmes, opérations et données en associant les données et les opérations qui les manipulent dans des objets, puis en forçant les programmes à manipuler les données d'un objet au travers des opérations de cet objet.
C'est cette approche qui est présentée dans la suite du présent document.



Bien évidemment, que ce soit dans l'approche par abstractions ou dans l'approche par objets, la modularité a un coût : la difficulté d'identifier les abstractions ou les classes d'objets, ainsi que leurs différentes relations. C'est pour cette raison qu'en amont de la phase de programmation sont apparues les méthodes d'analyse et/ou de conception orientées objet. La dernière en date, et à priori la plus complète, est la méthode UML (the Unified Modeling Language).

2.3 Langages orientés objet

Finalement, pour conclure cette introduction à la programmation orientée objet, voici une liste (non exhaustive) des langages de programmation orientés objet les plus utilisés :

– **SMALLTALK (1980)** : à l'origine des langages OO ; langage faiblement typé[4], interprété
– **C++ (1982)** : successeur du langage C ; langage fortement typé[5], hybride (POO et programmation structurée) ; un des plus utilisés à l'heure actuelle

[4]Dans un langage faiblement typé, le type (respectivement la classe) d'une variable (respectivement d'un objet) est déterminé lors de l'exécution. On parle également de typage dynamique.

[5]Dans un langage faiblement typé, le type (respectivement la classe) d'une variable (respectivement d'un objet) est déterminé lors de la compilation. Ainsi, durant l'exécution d'un programme, une variable contiendra toujours le même type d'objet. On parle également de typage statique.

– **EIFFEL (1988)** : langage fortement typé, pur OO ; concept de pré est post condition, d'invariant ; surtout utilisé en génie logiciel
– **JAVA (1995)** : langage fortement typé ; les sources sont d'abord compilé en bytecode, lequel est ensuite interprété par une machine virtuelle ; ce bytecode est indépendant de l'architecture matérielle et du système d'exploitation ; l'environnement Java de base (le JDK) intègre en standard de nombreuses bibliothèques de classes (interface graphique, réseau, processus, appel de fonctions distantes,...).

3 Objets

3.1 Définition

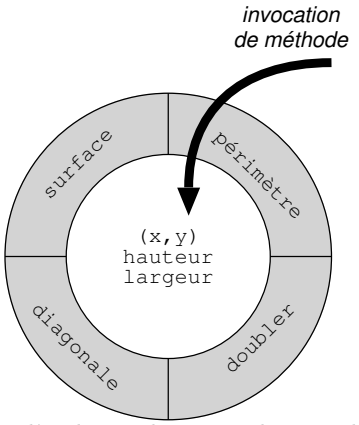
Un objet est une unité indépendante et distincte composée de méthodes et de fonctions. Les attributs décrivent l'**état interne** de l'objet (aspect statique). Les méthodes décrivent le **comportement** de l'objet (aspect dynamique), c'est-à-dire les opérations qui lui sont applicables ainsi que ses réactions aux événements du milieu dans lequel il se trouve.

3.2 Encapsulation

La programmation orientée objet répond au problème de la modularité par l'**encapsulation** des données et des opérations qui les manipulent dans des objets. L'encapsulation applique le principe d'**abstraction** : un objet n'est accessible que par ses opérations visibles (les méthodes de son interface externe) et son implémentation (les structures de données et attributs associés) est cachée. Ainsi les modifications sur les structures de données restent locales aux objets et sont sans effet sur les programmes qui utilisent ces objets. L'encapsulation apporte alors l'indépendance entre programmes, opérations et données. Un avantage immédiat est que différents programmes d'application peuvent partager les mêmes objets sans avoir à se connaître comme avec le mécanisme d'import/export (`#include`) de l'approche par abstraction.

Le principe d'encapsulation peut parfois être tempéré en autorisant un accès plus ou moins libre à certains attributs propres à un objet. Il faut pour cela être sûr de la stabilité dans le temps des choix d'implémentation de ces structures de données. Cependant, dès lors que l'implémentation des attributs est modifiée (par exemple en changeant leur type), il faut trouver et modifier tous les programmes utilisant ces attributs directement (sans passer par les méthodes de l'objet) comme avec l'approche par abstraction.

Prenons par exemple un objet rectangle. Celui-ci expose quatre méthodes : `surface` (calcule la surface du rectangle), `périmètre` (calcule le périmètre), `diagonale` (longueur d'une diagonale), `doubler` (multiplie par deux les dimensions du rectangle). Grâce au principe d'encapsulation (et d'abstraction de données), on pourrait très bien modifier la représentation interne du rectangle (i.e. remplacer les attributs `{(x,y),hauteur,largeur}` par `{(x1,y1),(x2,y2)}`) de manière complètement transparente par rapport aux programmes qui ne manipulent des rectangles qu'au travers de ces quatre méthodes.



L'encapsulation est donc une technique qui favorise la modularité, l'indépendance et la réutilisation des sous-systèmes en séparant l'interface d'un module (liste des services offerts) de son implémentation (structures de données et algorithmes).

4 Concepts de Base

4.1 Classes, attributs, méthodes

Plusieurs objets peuvent posséder une structure et des comportements en commun. Il devient alors utile de pouvoir les regrouper à l'aide d'un modèle général : une **classe**. Programmer revient alors à décrire des classes d'objets, à caractériser leur structure et leur comportement, puis à se servir de ces classes pour créer des objets (cf. instanciation).

Une classe est un moule pour fabriquer des objets (ou instances de classe) ayant la même structure et le même comportement.

La définition d'une classe est constituée par un ensemble d'**attributs** (également appelés **variables d'instances**) qui décrivent la structure des objets qui seront produits, et d'opérations, appelées **méthodes**, qui leur sont applicables.

Afin d'illustrer ces différentes notions, prenons un exemple très classique dans le domaine des objets : celui d'un compte en banque sur lequel on veut pouvoir effectuer des opérations de dépôt et de retrait d'argent. Chaque compte bancaire est bien caractérisé par deux attributs : ses valeurs de `crédit` et de `débit`. Du fait de l'encapsulation des attributs, le client d'une classe (i.e. l'utilisateur d'une classe) n'a pas accès directement à ces attributs. Pour modifier la valeur d'un attribut, il faut avoir prévu une méthode de modification. Un compte disposera donc également de deux méthodes : `déposer` pour créditer le compte d'une certaine somme, `retirer` pour débiter le compte. Finalement nous ajouterons une troisième méthode, `donnerSolde`, pour connaître le solde d'un compte bancaire.

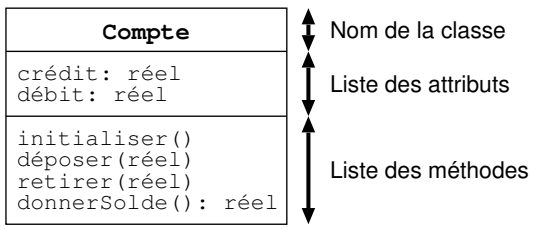
En programmation par objets, nous devons en premier lieu créer la classe `Compte`. Nous lui ajouterons ensuite ses attributs et ses méthodes. Du point de vue du développement, l'utilisateur d'une classe n'a en fait besoin de connaître que la liste des méthodes et (éventuellement) celle des attributs de cette classe : c'est la **spécification** de la classe (également appelée **interface**). Il n'a pas besoin de connaître le corps des méthodes. L'**implémentation** de la classe et de ses méthodes ne concernent que le concepteur/programmeur de la classe.

Dans la première partie de ce document consacrée à la programmation par objets en général, nous utiliserons un pseudo-langage objet pour définir les spécifications et implémentations de nos classes. L'utilisation d'un langage de programmation particulier, en l'occurrence Java, est reportée à la seconde partie. La spécification de la classe `Compte` peut alors s'écrire ainsi :

```
classe Compte: spécification
    attributs:
        crédit,débit: réel;

    méthodes:
        initialiser();
        déposer(réel);
        retirer(réel);
        donnerSolde(): réel;
fin Compte
```

Graphiquement, la classe `Compte` serait représentée de la manière suivante sur le graphe de classes :



Comme nous pouvons le constater, chaque classe dispose d'une méthode particulière : `initialiser`. Dans le cas présent, celle-ci ne prend aucun paramètre. Cette méthode sera automatiquement invoquée lors de la création d'un nouvel objet à partir de cette classe (cf. moule) pour initialiser ses attributs.

L'implémentation du corps des différentes méthodes de la classe `Compte` donnerait lieu aux algorithmes décrits ci-dessous. A savoir que dans la littérature la notation `Compte::retirer` est également utilisée pour désigner la méthode `retirer` de la classe `Compte`.

```
classe Compte: implémentation
    /* corps des différentes méthodes de la classe Compte */

    méthode initialiser()
        début
            crédit = 0 ;
            débit  = 0 ;
        fin

    méthode déposer(montant: réel)
        début
            crédit = crédit + montant ;
        fin

    méthode retirer(montant: réel)
        début
            débit = débit + montant ;
        fin

    méthode donnerSolde(): réel
        début
            retourner(crédit-débit);
        fin
fin Compte
```

L'une des caractéristiques notables des langages orientés objets est de pouvoir considérer n'importe quelle entité informatique comme un objet. Alors que la plupart des langages de programmation distinguent entre éléments simples (littéraux) et éléments composés (tableaux, structures, fichiers,...), la notion d'objet sert de concept fédérateur à tous les composants logiciels. Ainsi, même des unités simples comme les nombres peuvent être considérées comme des objets générés à partir de la classe `Nombre`. Il en est de même pour la classe `ChaîneDeCaractères` dont les objets seront eux-mêmes composés d'objets de la classe `Caractère`. Toutes les opérations manipulant des chaînes de caractères (comparaison, extraction de sous-chaîne, mise en majuscules,...) seraient alors définies comme étant des méthodes de la classe `ChaîneDeCaractères`.

4.2 Instances, messages

Par analogie avec les techniques de programmation structurée, une classe peut être vue comme un type, et les objets (ou instances) comme des variables d'un certain type.

Une **instance** est un objet généré à partir d'une classe existante (cf. moule) par un mécanisme appelé **instanciation** et matérialisé par l'opérateur `new`. Pour les personnes sachant déjà programmer en langage C, cet opérateur peut être vu comme un `malloc` : il alloue la mémoire, crée l'objet, puis retourne une **référence** ("pointeur") sur cet objet. Etant donné qu'en POO les objets ne sont connus qu'au travers de leurs références,

on utilise souvent, par abus de langage, le mot objet pour désigner une référence sur cet objet. Voici un exemple d'instanciation :

```
mon_compte : Compte;         /* mon_compte est une référence */
mon_compte = new Compte();   /* on crée une instance de la classe Compte */
```

Lors de l'instanciation d'un nouvel objet à partir d'une classe, il est nécessaire de fixer l'état initial de ce nouvel objet. A cet effet, à la création d'un objet, le mécanisme d'instanciation cherche (généralement automatiquement) une méthode particulière que nous noterons `initialiser` dans notre pseudo-langage. Si elle existe, elle est invoquée pour initialiser l'état de l'objet. C'est pour cette raison que cette méthode est appelée **constructeur** de l'objet. Dans l'exemple des comptes bancaires donné précédemment, dès qu'une instance de la classe `Compte` sera créée ses attributs `crédit` et `débit` seront donc immédiatement initialisés à `0`. Nous verrons par la suite qu'il est possible de définir plusieurs constructeurs pour une même classe selon les arguments que l'on désire passer à l'instanciation. A noter également qu'il existe de la même manière un **destructeur**, la méthode `détruire` (sans aucun paramètre) qui sera automatiquement invoquée lors de la destruction de l'objet.

⋆ ⋆ ⋆

En programmation orientée objets, les instances communiquent entre elles par **envoi de messages** (ou invocation de méthodes). C'est l'aspect dynamique des objets. Envoyer un message à un objet, c'est lui dire ce qu'il doit faire. Un message se compose de trois éléments : un **receveur** (l'objet qui reçoit le message), un **sélecteur** (nom de la méthode invoquée), et éventuellement d'un ou plusieurs **arguments**. La syntaxe d'un envoi de message est la suivante (deux notations possibles) :

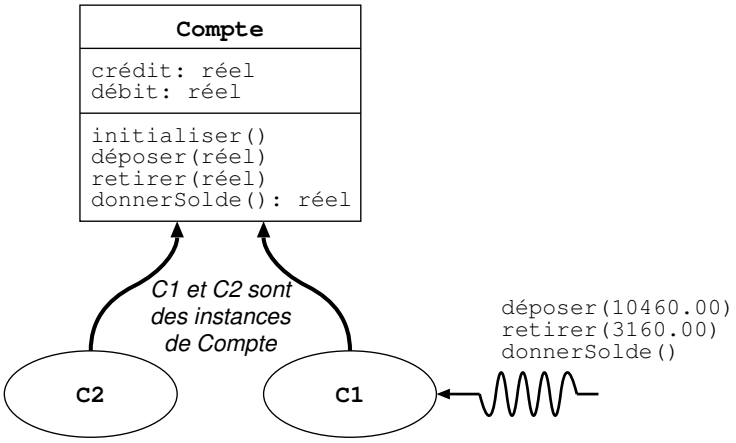
`<receveur>.<sélecteur>[(<liste d'arguments)]`
ou
`<receveur><=<sélecteur>[(<liste d'arguments)]`

Exemple de programme utilisant la classe `Compte` :

```
programme exempleCompte
début
    C1,C2 : Compte ;      /* 2 références sur des objets de la classe Compte */
    reste : réel   ;

    C1 = new Compte();    /* On instancie deux objets à partir de la classe Compte */
    C2 = new Compte();    /* Cela appelle automatiquement initialiser() sur ces objets */

    C1.déposer(10460.00);
    C1.retirer(3160.00);
    reste = C1.donnerSolde();
    afficher(reste);
fin
```



Les objets s'envoient parfois des messages à eux-mêmes. En effet, il est parfois indispensable de faire appel à une autre méthode définie dans la même classe, et donc de renvoyer un message à l'objet qui vient juste d'en recevoir un. Il existe pour cela une variable particulière, généralement appelée `self`, qui représente le receveur du message. `self` représente le "je" du langage ordinaire, la possibilité de représenter des actions d'un point de vue subjectif. En d'autres termes, l'objet invoque une de ses propres méthodes.

⋆ ⋆ ⋆

Comme on peut s'en douter, la plupart des méthodes ne se résument pas à effectuer des calculs arithmétiques. Les traitements qu'elles réalisent peuvent les amener à envoyer, à leur tour, de nouveaux messages vers des objets dont les références sont connues au travers d'attributs, de variables locales ou globales, ou ont tout simplement été passées en arguments. Prenons comme exemple la classe `Banque` qui va stocker tous les comptes bancaires (instances de `Compte`) dans un attribut de classe `Dictionnaire`. Cette dernière dispose, entre autres, des méthodes `addElement` (création d'une nouvelle entrée à partir d'un compte et d'un index) et `elementAt` (retourne l'objet `Compte` associé à un index donné).

```
classe Banque: spécification
    attributs:
        comptes: Dictionnaire;
    méthodes:
        initialiser();
        ouvrirCompte(numCompte);
        déposer(numCompte,réel);
        retirer(numCompte,réel);
        transférer(numCompte,numCompte,réel);
        donnerSolde(numCompte): réel;
fin Banque
```

Dans l'implémentation des différentes méthodes de la classe `Banque`, nous allons nous servir des références objet renvoyées par la méthode `elementAt` de la classe `Dictionnaire` comme receveur pour envoyer de nouveaux messages. On parle alors de **composition de messages**.

```
classe Banque: implémentation
    méthode initialiser()
        début
            comptes = new Dictionnaire();
        fin
```

```
    méthode ouvrirCompte(index: numCompte)
        début
            comptes.addElement(index,new Compte);
        fin

    méthode déposer(index: numCompte, montant: réel)
        début
            comptes.elementAt(index).déposer(montant);
        fin

    méthode retirer(index: numCompte, montant: réel)
        début
            comptes.elementAt(index).retirer(montant);
        fin

    méthode transférer(from: numCompte, to: numCompte, montant: réel)
        début
            self.retirer(from,montant);
            self.déposer(to,montant);
        fin

    méthode donnerSolde(index: numCompte): réel
        début
            retourner(comptes.elementAt(index).donnerSolde());
        fin
fin Banque
```

5 Concepts Avancés

5.1 Héritage

Une autre particularité notable dans l'organisation des classes en programmation orientée objets est l'héritage de propriétés.

L'**héritage** est un mécanisme permettant à une nouvelle classe de posséder automatiquement les variables et les méthodes de la classe dont elle **dérive**.

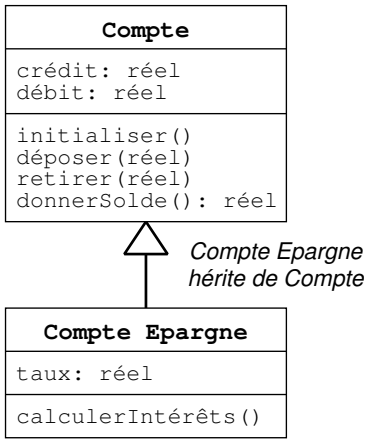
Supposons que nous soyons en train de développer un système de fenêtrage. Nous avons besoin de fenêtres différentes pour l'édition d'un texte, l'évaluation d'une commande, ou encore la visualisation d'un graphique. Bien que fournissant des fonctionnalités différentes, ces fenêtres disposent toutefois d'un certain nombre de caractéristiques communes : chacune possède un cadre, est positionnée à un certain endroit de l'écran, est capable de réagir aux événements extérieur tels qu'un click de souris, peut être déplacée et redimensionnée,... Il serait dommage (et surtout fastidieux) de devoir tout réécrire à chaque fois que l'on définit un nouveau type de fenêtre. Il serait préférable de pouvoir factoriser toutes ces informations et traitements communs à l'intérieur d'une même classe. C'est précisément ce que permet de réaliser l'héritage, mécanisme de description très puissant que l'on ne rencontre guère que dans les approches orientées objets (programmation, langages, conception,...) et qui permet de décrire des structures génériques ou abstraites. Toute classe peut engendrer d'autres classes, appelées **sous-classes** de la première. Par défaut, une sous-classe possède (hérite de) l'ensemble des caractéristiques (attributs et méthodes) de sa **super-classe**, ces caractéristiques pouvant éventuellement être ensuite modifiées au niveau de la sous-classe. A noter qu'en tant que classe, une sous-classe peut à son tour posséder différentes sous-classes, et ainsi de suite. La propriété d'héritage est

transitive.

Dans notre exemple, il est possible de créer une classe `Fenêtre` générale qui contient tout ce qui est commun à tous les différents types de fenêtres, et d'en dériver ensuite des classes de fenêtres particulières telles que `FenêtreEdition`, `FenêtreEvaluation` et `FenêtreGraphique`. A son tour, la classe FenêtreEdition pourrait alors être dérivée en fenêtres d'édition particulières telles que `FenêtreEditionWord`, `FenêtreEditionLaTeX`,... Autre exemple, pour créer une classe représentant un compte épargne, c'est-à-dire un compte qui produit des intérêts qui se cumulent au crédit du compte, il n'est pas nécessaire de redéfinir toutes les caractéristiques déjà présentes dans la classe `Compte`, mais seulement de caractériser les différences qui existent avec cette dernière. Ces différences sont au nombre de deux : un nouvel attribut `taux` et une nouvelle méthode `calculerIntérêts`. La classe `CompteEpargne` peut être définie ainsi :

```
classe CompteEpargne: spécification
    sous-classe de Compte
    attributs:
        taux: réel;
    méthodes:
        calculerIntérêts();
fin CompteEpargne
```

Au niveau du graphe de classes, l'héritage entre classes se représente de la manière suivante :



Maintenant, toutes les instances de `CompteEpargne` disposent implicitement des attributs `crédit` et `débit`, mais savent également répondre aux messages `déposer`, `retirer` et `donnerSolde`. Toutes ces caractéristiques sont héritées de la classe `Compte`.

```
classe CompteEpargne: implémentation
    méthode calculerIntérêts()
        début
            crédit = crédit + ((self.donnerSolde() * taux) ;
        fin
fin CompteEpargne
```

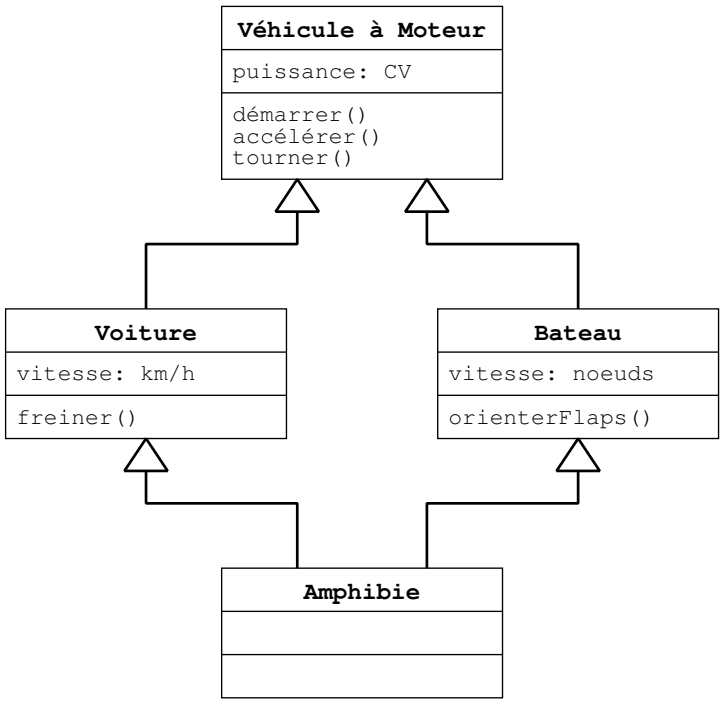
L'héritage permet de répondre à différents besoins :

– Une classe existante `A` est incomplète pour les besoins de l'application. On définit une nouvelle classe `B` qui hérite de `A` et on y ajoute les nouvelles fonctionnalités (attributs et méthodes) nécessaires. On étend les fonctionnalités de la classe existante : **enrichissement** d'attributs et de méthodes.

– Une classe existante `A` est suffisante pour répondre aux besoins, mais son comportement dans certains cas de figure n'est pas totalement satisfaisant. On définit une nouvelle classe `B` qui hérite de `A` et on y redéfinit les méthodes de `A` qui ne sont pas satisfaisantes. On parle de **substitution** : les attributs et les méthodes définis dans la sous-classe `B` sont prioritaires par rapport aux attributs et aux méthodes de même nom définis dans la classe `A`. Attention toutefois, car s'il est possible au niveau de `B` de changer l'implémentation d'une méthode héritée de `A`, il n'est pas permis de modifier sa signature (nom, type et nombre des arguments, type de la valeur de retour).

Le mécanisme d'héritage induit un nouveau style de programmation qui procède par affinages successifs de classes prédéfinies. Créer un programme dans ces conditions consiste à définir des sous-classes de classes plus générales, c'est-à-dire à augmenter les caractéristiques du langage en décrivant des classes plus spécifiques, mieux adaptées au problème à résoudre.

Pour conclure cette section sur l'héritage, il reste encore à préciser que l'on peut avoir de l'**héritage simple** (une sous-classe n'hérite que d'une seule classe) ou de l'**héritage multiple** (une sous-classe hérite de plusieurs classes), ce dernier n'étant pas forcément supporté par tous les langages orientés objets (ou sous des formes légèrement différentes).



Sur ce graphe de classes, `Voiture` et `Bateau` héritent tous deux de la classe `Véhicule` (héritage simple), et la classe `Amphibie` hérite à la fois de la classe `Voiture` et de la classe `Bateau` (héritage multiple). Comme nous pouvons le constater, l'héritage multiple peut introduire des **ambiguïtés** :

– La classe `Amphibie` hérite de la méthode `démarrer` définie dans la classe `Véhicule`, à la fois au travers de la classe `Voiture` et de la classe `Bateau`. Si aucune des deux classes `Voiture` ou `Bateau` n'a modifié cette méthode, peu importe la manière dont on accède à la méthode `démarrer` : ce sera la même. Mais prenons maintenant le cas de la méthode `tourner`. Celle-ci sera naturellement redéfinie dans la classe `Voiture` (pour tourner les roues) et dans la classe `Bateau` (pour orienter le gouvernail). Mais si nous invoquons la méthode `tourner` sur un objet de la classe `Amphibie` (car en tant que spécialisation de la classe `Véhicule`, la classe `Amphibie` possède une méthode `tourner`), laquelle de ces deux versions sera prise en compte ?

– Plus subtile encore, le cas de l'attribut `vitesse`. Celui-ci n'existe pas au niveau de la classe `Véhicule`. Par contre, les classes `Voiture` et `Bateau` le définissent chacune de leur côté, mais avec un type différent.

Etant donné que la classe `Amphibie` hérite de ces deux classes, quel sera le type de l'attribut `vitesse` au niveau de la classe `Amphibie` ?

Difficile à priori de répondre à ces questions. Certains langages ne lèvent pas les ambiguïtés, d'autres spécifient l'ordre du graphe de classe et définissent des priorités. Mais bien évidemment, du point de vue de la certification des programmes, il est préférable de ne pas laisser la résolution de ces ambiguïtés "au hasard" et de redéfinir explicitement le type de l'attribut `vitesse` et le corps de la méthode `tourner` au niveau de la classe `Amphibie`. Il suffit simplement d'y penser...

5.2 Polymorphisme, liaison dynamique

On dit d'une méthode qu'elle est **polymorphe** si elle réalise des actions différentes selon la nature des objets sur lesquels elle s'applique. On distingue trois formes de polymorphisme de méthode : **surcharge**, **méthodes prédominantes** et **méthodes abstraites** (ou virtuelles).

⋆ ⋆ ⋆

Une méthode est dite surchargée s'il existe plusieurs implémentations associées à son nom. C'est le cas par exemple de la méthode `modifierCoordonnées` de la classe `Vecteur3D` définie ci-dessous.

```
classe Vecteur3D: spécification
    attributs:
        x,y,z: réel;
    méthodes:
        modifierCoordonnées(réel,réel,réel);
        modifierCoordonnées(Vecteur3D);
        modifierCoordonnées(tableau[1..3] de réel);
fin Vecteur3D
```

Comme cela est illustré par le programme `clientV3D`, la méthode `modifierCoordonnées` peut être invoquée en passant comme arguments du message soit trois réels, soit un autre vecteur 3D, soit un tableau de trois réels.

```
programme clientV3D
début
    t  : tableau[1..3] de réel = {5.5,10,3.2} ;
    v1 : Vecteur3D = new Vecteur3D() ;
    v2 : Vecteur3D = new Vecteur3D() ;
    v3 : Vecteur3D = new Vecteur3D() ;

    v1.modifierCoordonnées(10,11.5,15.8);
    v2.modifierCoordonnées(v1);
    v3.modifierCoordonnées(t);
fin
```

Avec cette forme de polymorphisme, c'est le compilateur qui détermine l'implémentation de la méthode `modifierCoordonnées` à invoquer en fonction des arguments utilisés. Etant donné que le type (ou la classe) des arguments sont connus lors de la compilation, on parle de **liaison statique**.

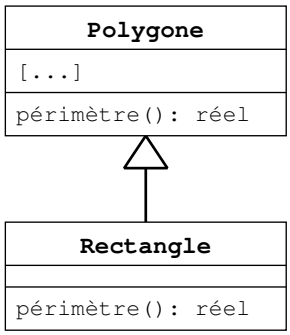
⋆ ⋆ ⋆

La prédominance de méthode est quant à elle introduite par l'héritage du fait de la redéfinition de méthodes. Prenons par exemple le cas d'une classe `A` dont la méthode `m` ne nous convient pas. Nous développons alors une nouvelle classe `B` qui hérite de `A` et qui redéfinit la méthode `m` (avec des paramètres identiques).

Dans ce cas, lors de l'invocation de la méthode `m` sur une instance de la classe `B`, c'est la méthode `m` de `B` qui prédomine par rapport à celle de la classe `A`.

Une méthode d'une classe peut être invoquée par toute instance de cette classe mais aussi par toute instance de ses sous-classes. Un même message envoyé à plusieurs objets pourra ainsi être interprété différemment suivant le comportement des objets qui le reçoivent (i.e. s'ils ont ou non redéfini cette méthode).

Par exemple, pour calculer le périmètre d'un rectangle (sous-classe de polygone), on ne va pas se servir de la méthode **`périmètre`** de la classe `Polygone` (somme de la longueur de tous ses côtés) car celle de la classe `Rectangle` est plus rapide (deux fois la somme de la largeur et de la longueur).

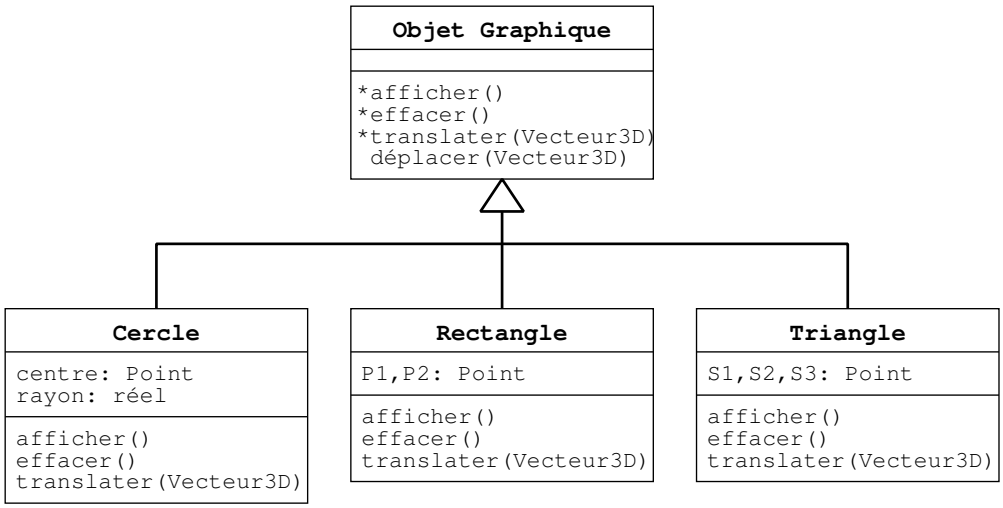


La détermination de la méthode à invoquer suite à un envoi de message aura donc lieu à l'exécution (en fonction de la classe réelle de l'objet receveur, i.e. instancié à partir de `Polygone` ou de `Rectangle`) et non à la compilation. On parle alors de **liaison dynamique**.

⋆ ⋆ ⋆

L'utilisation conjointe de l'héritage et de la variable `self` introduit une nouvelle forme de polymorphisme de méthode, complètement spécifique à la programmation par objets. Ce mécanisme très puissant consiste à pouvoir définir des méthodes abstraites (également dites virtuelles) indépendantes de leur réalisation dans une sous-classe particulière.

Ainsi, supposons que l'on veuille faire un logiciel de visualisation d'objets graphiques. Tous ces objets savent s'afficher, s'effacer et se translater (modifier ses coordonnées). Mais les procédures d'affichage et d'effacement sont différentes selon la nature des entités graphiques (cercle, rectangle, triangle,...). En revanche, il est possible d'accorder tout le monde sur le fait qu'un objet graphique, pour se déplacer, doit d'abord s'effacer, puis modifier ses coordonnées, et enfin se réafficher.



La classe `ObjetGraphique` définit quatre méthodes (`dessiner`, `effacer`, `translater`, `déplacer`) dont trois sont abstraites (`dessiner`, `effacer` et `translater`). Cela signifie que ces trois méthodes **ne seront pas implémentées** au niveau de la classe `ObjetGraphique` elle-même, mais **devront obligatoirement l'être** dans ses sous-classes. Il sera donc possible d'invoquer la méthode dessiner sur des objets des classes `Cercle`, `Rectangle` ou `Triangle` (le corps de la méthode aura été implémenté), mais pas sur des objets de la classe `ObjetGraphique` (corps non défini). La classe `ObjetGraphique` est appelé **classe abstraite** car certaines de ses méthodes ne sont pas implémentées. D'ailleurs, certains langages refusent tout simplement d'instancier des objets à partir d'une classe abstraite. Afin de distinguer les méthodes abstraites des méthodes

"classiques", nous ajouterons le symbole `'*'` devant leur nom sur le graphe de classes.

La définition de la méthode `déplacer` est alors totalement générique, car elle ne présage aucunement de la manière dont les différents objets graphiques vont s'afficher, s'effacer ou se translater. Elle peut donc directement implémentée au niveau de la classe `ObjetGraphique`.

```
classe ObjetGraphique: implémentation
    méthode déplacer(v : Vecteur3D)
        début
            self.effacer();
            self.translater(v);
            self.afficher();
        fin
fin ObjetGraphique
```

Elle suppose simplement que ces différents objets savent chacun répondre aux trois message `effacer`, `translater` et `afficher`. De ce fait, si l'on désire ajouter un nouveau type d'objet graphique (polygone, ellipse,...), il suffira de vérifier qu'ils savent répondre convenablement à ces trois messages pour qu'ils disposent automatiquement de toute une panoplie de comportements plus complexes tels que le déplacement à l'écran. Ces méthodes génériques sont très utilisées dans la pratique et sont la clef du développement des interfaces graphiques actuelles.

5.3 Généricité

Le concept de polymorphisme peut également s'appliquer aux classes elles-mêmes. On parle alors de **généricité** (polymorphisme de classe). Il s'agit essentiellement de classes décrivant des structures de données standard telles que les listes, les piles, les arbres, les graphes,...Les classes génériques implémentent des méthodes indépendamment des objets qu'elles contiennent.

```
classe Liste<T>    /* T est le paramètre de généricité */
                   /* les éléments de la liste seront de type T */
    méthodes:
        longueur(): entier;
        ième(entier): T;
        insérerEnQueue(T);
        supprimer(entier);
fin Liste
```

Par la suite, si l'on souhaite instancier des listes dans un programme client, il faudra préciser le type des paramètres de généricité :

- Une liste d'entiers : `l1 : new Liste<entier>() ;`
- Une liste d'objets graphiques : `l2 : new Liste<ObjetGraphique>() ;`
- Une liste de comptes bancaires : `l3 : new Liste<Compte>() ;`

Nous n'irons pas plus loin en ce qui concerne la généricité car peu de langages supportent ce concept, et quand c'est le cas, la mise en œuvre de la généricité est souvent "propriétaire".

Index