Chapitre 2

Une CLASSE : pour définir de nouveaux objets

Le paradigme de la POO est que c'est une programmation dirigée par les données, et non plus une programmation dirigée par les traitements, comme c'était le cas avec des langages tels que C, Pascal, ou encore le langage Java tel qu'il a été pratiqué en première année.

L'approche algorithmique que nous avons vue jusqu'à présent consiste à décrire des (sous)-algorithmes, c'est à dire des actions ou des fonctions, qui agissent sur des variables situées « à l'extérieur », c'est à dire non liées par une structure, au code de ces actions ou de ces fonctions.

Exemple. On considère un programme qui manipule trois variables entières h, m et s pour définir un horaire, et une fonction qui détermine si l'horaire défini par h, m et s est un horaire valide (i.e. $0 \leq h < 24$, $0 \leq m < 60$, $0 \leq s < 60$).

Les variables seront déclarées :

```
entier h, m, s
```

puis ailleurs la fonction

```
booléen horaireCorrect(h,m,s)
données entier h, m, s
        retourner 0<=h et h<24 et 0<=m et m<60 et 0<=s et s<60
fin
```

Aucune structure ne matérialise le fait que h, m, s et la fonction horaireCorrect() sont liées au même concept, celui d'*horaire*.

On pourrait rassembler h, m et s dans une structure, nommée Horaire, mais le traitement réalisé par horaireCorrect() resterait séparé de la donnée décrite par Horaire.

```
Structure Horaire
        entier h, m, s
fin Structure
```

puis

```
booléen horaireCorrect(h,m,s)
données entier h, m, s
        retourner 0<=h et h<24 et 0<=m et m<60 et 0<=s et s<60
fin
```

En POO, on va se concentrer d'abord sur les entités que l'on va manipuler avant de se concentrer sur la façon dont on va les manipuler. Cela s'obtiendra par *encapsulation* du code et des méthodes. Cela signifie que l'on va regrouper dans une même structure les données et les actions ou fonctions qui les manipulent.

Cette structure s'appelle une *classe*. Les données sont appelées les *attributs*. Les actions ou fonctions sont appelées *méthodes*.

La définition de la classe ressemble maintenant à :

```
classe Horaire {
        entier h, m, s ; // attributs

        // méthode
        booléen horaireCorrect(h,m,s)
                données entier h, m, s
                        retourner 0<=h et h<24 et 0<=m et m<60 et 0<=s et s<60 ;
                fin fonction
} // fin de la classe
```

On remarque tout de suite que puisque les données h, m, s et la fonction horaireCorrect() sont maintenant réunies au sein d'une même entité (la classe), on n'a plus besoin de communiquer h, m et s en paramètres à horaireCorrect(). Celle-ci les connaît déjà du fait qu'ils appartiennent à la même classe.

1 Vue d'ensemble et définition d'une classe

1.1 Similarités avec la notion de Structure

Concrètement, une classe peut être vue comme une extension de la notion de *structure* vue en L1.

Rappel. Les structures permettent de rassembler en une seule entité plusieurs éléments. En effet, bien des objets (réels ou virtuels) ne sont pas représentables par la donnée d'une seule grandeur. Il est naturel de vouloir mettre ensemble toutes les grandeurs qui permettent de décrire un objet particulier.

Par exemple, pour décrire un nombre entier, on a besoin d'une seule grandeur : ce nombre ! Un entier n suffit. Mais pour décrire un nombre complexe, un seul nombre ne suffit pas : il faut deux réels, la partie imaginaire et la partie réelle.

Une structure permet de rassembler au sein d'une même entité des variables diverses qui caractérisent un « objet » particulier.

Exemple.
- Nombre Complexe : partie réelle, partie imaginaire
- Carte à jouer : valeur, couleur
- Point dans le plan : coordonnée en x, coordonnée en y
- Point dans l'espace : coordonnée en x, coordonnée en y, coordonnée en z

On a donc rassemblé l'ensemble des données qui caractérisent une entité particulière, mais on ne dit rien sur la façon dont ces données doivent être traitées. Une classe est une structure étendue, qui, à la description des données, associe aussi les *traitements* autorisés sur ces données.

1.2 Définition d'une classe

Définition 2.1 (Classe)**.** *Une classe est la description d'une famille d'objets ayant une même structure et un même comportement. Elle est caractérisée par :*
- *un* nom*,*
- *une composante statique : des* attributs*, qui sont des variables typées ;*
- *une composante dynamique : des* méthodes *définissant le comportement des objets de cette classe ; ce sont des fonctions.*

Les attributs caractérisent l'état des objets pendant l'exécution du programme.

Les méthodes manipulent les attributs des objets et caractérisent les manipulations autorisées de ces objets.

1.3 Différence entre classe et objet

Une classe constitue la définition de toute une famille d'objets. Un objet est un exemplaire en particulier d'une entité dont la structure est conforme à celle décrite par la classe.

2 Un premier exemple : un nombre complexe

Considérons des nombres complexes, et décrivons les par une classe. On l'a dit, il faut deux réels pour décrire un nombre complexe : sa partie réelle (notons la pReelle) et sa partie imaginaire (notons la pImag). Parmi les opérations associées à un nombre complexe, on peut calculer son module ($\sqrt{pReelle^2 + pImag^2}$), ou son conjugué (changement de signe de la partie imaginaire : $-pImag$).

N.B. les opérations d'addition et de multiplication sont laissées à titre d'exercice.

On peut décrire un nombre complexe par une classe. Le nom de cette classe sera naturellement NombreComplexe. La classe possédera deux attributs de type réel : pReelle et pImag. Elle possédera deux méthodes : module() et conjugue(). La méthode module() ne change pas l'état du nombre complexe mais renvoie un résultat réel (le module), tandis que la méthode conjugue() ne renvoie aucun résultat mais change l'état du nombre, en inversant le signe de sa partie imaginaire.

Représentations d'une classe. On emploiera la notation objet standard UML (*Unified Modelling Language*) pour représenter une classe de manière synthétique :

NombreComplexe
réel pReelle, pImag
réel module() *rien* conjugue()

Ceci constitue un *modèle* de la classe NombreComplexe. C'est une vue synthétique du contenu de la classe. Un modèle permet à un utilisateur de la classe de voir comment il peut utiliser la classe, sans qu'il ait besoin de savoir comment elle a été programmée. Comme ce modèle ne dit rien sur le code des méthodes, il ne constitue pas une définition complète.

On définit à proprement parler une classe par son code, décrit en PDL++ pour le cours et les TDs.

```
classe NombreComplexe {
  réel pReelle, pImag ;

  réel module()
        retourner racine(pReelle*pReelle + pImag*pImag) ;
  fin
  rien conjugue()
        pImag ← -pImag ;
  fin
}
```

En TP, on utilisera le langage Java, ce qui donne une troisième représentation.

```
class NombreComplexe {
      double pReelle, pImag ;

      double module() {
            return Math.sqrt(pReelle*pReelle + pImag*pImag) ;
      }

      void conjugue() {
            pImag = -pImag) ;
      }
}
```

Ce code Java sera enregistré dans un fichier nommé impérativement NombreComplexe.java. Le programme principal qui va utiliser des objets de cette classe sera décrit séparément, dans un autre fichier.

Tout ce qui se situe entre les deux accolades d'ouverture et de fermeture de la classe s'appellera *l'intérieur* de la classe, tandis que les autres parties d'un programme constitueront *l'extérieur*, ou encore *l'environnement*.

Une classe constitue une définition de type. On peut donc déclarer (dans un programme principal par exemple), une variable c de type NombreComplexe :

```
NombreComplexe c ;
```

Un tel type est appelé un *type objet*.

3 Formalisation des attributs et des méthodes

3.1 Attributs

Les attributs d'une classe sont les « grandeurs » qui caractérisent l'objet que l'on définit. Plus précisément, ils sont définis comme des *variables typées*, dont les valeurs définissent *l'état* de l'objet. Un changement d'état de l'objet se traduit par un changement de valeur d'au moins un des attributs.

Syntaxiquement, on déclare un attribut d'une classe comme une variable typée, mais *à l'intérieur* de la classe. Les attributs sont déclarés à l'extérieur des méthodes, en général juste avant celles-ci.

N.B. Une variable typée déclarée à l'intérieur d'une méthode n'est pas un attribut de la classe mais une variable locale à la méthode.

N.B. Le type de l'attribut peut être un type dit *primitif* (entier, réel, caractère, booléen), mais il peut être d'un type plus compliqué (tableau, ou type objet.

3.2 Méthodes

Les méthodes dans une classe servent à :
- modifier l'état de l'objet, i.e. changer la valeur d'au moins un des attributs (exemple : conjugue())
- renseigner sur l'état de l'objet (exemple : module())

3.2.1 Syntaxe

Une méthode se déclare de la même manière qu'une action ou qu'une fonction :

type_de_retour nom_methode(parametres_de_la_methode) ← en-tête de la méthode

suivi de son code *(le corps de la méthode)*, mais elle est déclarée *à l'intérieur* d'une classe (en général à la suite des attributs)

N.B. Dans un modèle UML, seule l'en-tête de la méthode apparaît (pas le corps).

Langage Java.

```
type_retour nom_methode(type_param_1 nom_param_1, ..., type_param_n nom_param_n)   en-tête de la méthode
{
... suite d'instructions java ...                                                  corps de la méthode
}
```

3.2.2 Passage des paramètres aux méthodes

Commençons par souligner un point fondamental : les attributs d'une classe sont automatiquement connus de toutes les méthodes de la classe. Cela signifie qu'ils ne doivent *jamais* être passés en paramètre aux méthodes. Les paramètres servent uniquement de points de communication entre la méthode et l'extérieur de la classe : on a dit qu'une méthode pouvait servir à changer l'état d'un objet. Ce changement peut se faire en fonction de valeurs communiquées via les paramètres aux méthodes.

Par exemple, pour utiliser un objet « bouton » dans une interface graphique, on devra communiquer en paramètre le texte à afficher dans le bouton (par exemple Quitter, ou Sauvegarder, ...) Autre exemple, pour multiplier un nombre complexe par un autre, cet autre nombre doit être reçu en paramètre.

Java n'implémente qu'un seul mode de passage des paramètres à une méthode : le passage *par valeur*. En conséquence, la valeur de l'argument passé à une méthode ne peut pas être modifié. Notons cependant que ce discours sera nuancé quand nous aurons étudié la notion de référence.

Nous adopterons en langage algorithmique le même fonctionnement qu'en Java, c'est à dire le passage par valeur uniquement. On ne prendra donc pas la peine de préciser qu'un paramètre est une donnée, un résultat ou encore une donnée-résultat. Tous les paramètres seront des données.

3.2.3 Surcharge des méthodes

Il est possible de donner plusieurs définitions d'une même méthode dans une classe, à condition que les paramètres diffèrent d'une définition à l'autre.

Par exemple, pour une méthode qui permettrait de dessiner un cercle à l'écran, on peut imaginer une première version avec comme seul paramètre le rayon du cercle à dessiner (le cercle est alors dessiné au milieu de l'écran), et une autre version avec comme paramètres

le rayon et la position du centre. Ces deux méthodes porteront le même nom (par exemple dessinerCercle(...)).

Ce qui permet de distinguer les deux méthodes, c'est qu'elles n'ont pas la même liste de paramètres.

Définition 2.2 (Signature)**.** *La signature d'une méthode est définie par son nom associé à la liste des* types *de ses paramètres, dans l'ordre où ils apparaissent.*

N.B. Le type de retour d'une méthode ne fait pas partie de sa signature.

Exemple. Soit la méthode
entier toto(Chaîne ch, NombreComplexe c, réel x)

Sa signature est
toto(Chaîne, NombreComplexe, réel)

On peut donner dans une classe autant de définitions d'une même méthode qu'on le souhaite, à condition que chaque définition ait une signature distincte de celles des autres.

On dit dans ce cas qu'on *surcharge* la méthode.

Définition 2.3 (Surcharge)**.** *On dit qu'une méthode en* surcharge *une autre de la même classe quand elle a le même nom mais une signature différente.*

Exemple. entier toto(entier n1, entier n2) surcharge la méthode toto(...) vue plus haut.