Chapitre 1

LE CONCEPT D'OBJET

Apparue au début des années 70, la programmation orientée objet répond aux nécessités de l'informatique professionnelle. Elle offre aux concepteurs de logiciels une grande souplesse de travail, permet une maintenance et une évolution plus aisée des produits.

Mais sa pratique passe par une approche radicalement différente des méthodes de programmation traditionnelles : avec les langages à objets, le programmeur devient metteur en scène d'un jeu collectif où chaque objet-acteur se voit attribuer un rôle bien précis.

Ce cours a pour but d'expliquer les règles de ce jeu. La syntaxe de base du langage C++, exposée dans un précédent cours, est supposée connue.

1.1 Objet usuel

(1.1.1) Comment décrire un objet usuel ? Prenons exemple sur la notice d'utilisation d'un appareil ménager. Cette notice a généralement trois parties :

a. une description physique de l'appareil et de ses principaux éléments (boutons, voyants lumineux, cadrans etc.), schémas à l'appui,
b. une description des fonctions de chaque élément,
c. un mode d'emploi décrivant la succession des manœuvres à faire pour utiliser l'appareil.

Seules les parties *a* et *b* sont intrinsèques à l'appareil : la partie *c* concerne l'utilisateur et rien n'empêche celui-ci de se servir de l'appareil d'une autre manière, ou à d'autres fins que celles prévues par le constructeur.

Nous retiendrons donc que pour décrire un objet usuel, il faut décrire ses composants, à savoir :

1. *les différents éléments qui le constituent,*
2. *les différentes fonctions associées à ces éléments.*

(1.1.2) Les éléments qui constituent l'objet définissent à chaque instant l'*état* de l'objet — on peut dire : son aspect spatial. Les fonctions, quant à elles, définissent le *comportement* de l'objet au cours du temps.

Les éléments qui constituent l'objet peuvent se modifier au cours du temps (par exemple, le voyant d'une cafetière peut être allumé ou éteint). Un objet peut ainsi avoir plusieurs états. Le nombre d'états possibles d'un objet donne une idée de sa *complexité*.

(1.1.3) Pour identifier les composants d'un objet usuel, une bonne méthode consiste à faire de cet objet une description littérale, puis de souligner les principaux noms communs et verbes. Les noms communs donnent les éléments constituants, les verbes donnent les fonctions.

Illustrons cette méthode dans le cas d'un objet très simple, un marteau :

On peut en faire la description suivante :

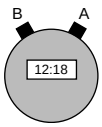
> "Ce marteau comporte un manche en bois, une extrémité plate en métal et une extrémité incurvée également en métal. Le manche permet de saisir le marteau, l'extrémité plate permet de frapper quelque chose et l'extrémité incurvée permet d'arracher quelque chose."

D'où la *fiche descriptive* :

nom : marteau	
éléments :	*fonctions :*
manche	saisir
extrémité plate	frapper
extrémité incurvée	arracher

Pour vérifier que nous n'avons rien oublié d'important dans une telle fiche descriptive, il faut imaginer l'objet à l'œuvre dans une petite scène. Le déroulement de l'action peut alors révéler des composants qui nous auraient échappé en première analyse (dans notre exemple, quatre acteurs : un marteau, un clou, un mur et un individu ; pour planter le clou dans le mur, l'individu saisit le marteau par son manche, puis frappe sur le clou avec l'extrémité plate ; il s'aperçoit alors que le clou est mal placé, et l'arrache avec l'extrémité incurvée).

(1.1.4) Prenons comme deuxième exemple un chronomètre digital :



> "Ce chronomètre comporte un temps qui s'affiche et deux boutons A et B. Quand on presse sur A, on déclenche le chronomètre, ou bien on l'arrête. Quand on presse sur B, on remet à zéro le chronomètre."

D'où la *fiche descriptive* :

nom : chronomètre	
éléments :	*fonctions :*
boutons A, B	afficher
temps	presser sur un bouton
	déclencher
	arrêter
	remettre à zéro

Remarquons que l'utilisateur ne peut pas modifier directement le temps affiché : il n'a accès à ce temps que de manière indirecte, par l'intermédiaire des fonctions de l'objet. Cette notion d'accès indirect jouera un rôle important dans la suite (1.3).

1.2 Objet informatique — Classe

(1.2.1) L'ordinateur est un appareil possédant une très grande complexité — liée à un très grand nombre d'états — et un comportement très varié — lié à la façon dont on le programme. Il nous servira d'*objet universel* capable de simuler la plupart des objets usuels.

(1.2.2) Programmer un ordinateur, c'est lui fournir une série d'instructions qu'il doit exécuter. Un langage de programmation évolué doit simplifier le travail du programmeur en lui offrant la possibilité :

– d'écrire son programme sous forme de petits modules autonomes,
– de corriger et faire évoluer son programme avec un minimum de retouches,
– d'utiliser des modules tout faits et fiables.

De ce point de vue, les langages à objets comme le C++ sont supérieurs aux langages classiques comme le C, car ils font reposer le gros du travail sur des "briques logicielles intelligentes" : les objets. Un programme n'est alors qu'une collection d'objets mis ensemble par le programmeur et qui coopèrent, un peu comme les joueurs d'une équipe de football supervisés par leur entraîneur.

(1.2.3) Transposé en langage informatique, (1.1.1) donne :

Un objet est une structure informatique regroupant :
– *des variables, caractérisant l'état de l'objet,*
– *des fonctions, caractérisant le comportement de l'objet.*

Les variables (resp. fonctions) s'appellent *données-membres* (resp. *fonctions-membres* ou encore *méthodes*) de l'objet. L'originalité dans la notion d'objet, c'est que variables et fonctions sont regroupées dans une même structure.

(1.2.4) *Un ensemble d'objets de même type s'appelle une classe.*

Tout objet appartient à une classe, on dit aussi qu'il est une *instance* de cette classe. Par exemple, si l'on dispose de plusieurs chronomètres analogues à celui décrit en (1.1.4), ces chronomètres appartiennent tous à une même classe "chronomètre", chacun est une instance de cette classe. En décrivant la classe "chronomètre", on décrit la structure commune à tous les objets appartenant à cette classe.

(1.2.5) *Pour utiliser les objets, il faut d'abord décrire les classes auxquelles ces objets appartiennent.*

La description d'une classe comporte deux parties :
– une partie *déclaration*, fiche descriptive des données et fonctions-membres des objets de cette classe, qui servira d'interface avec le monde extérieur,
– une partie *implémentation*, contenant la programmation des fonctions-membres.

1.3 Encapsulation

(1.3.1) Dans la déclaration d'une classe, il est possible de protéger certaines données-membres ou fonctions-membres en les rendant invisibles de l'extérieur : c'est ce qu'on appelle l'*encapsulation*.

A quoi cela sert-il ? Supposons qu'on veuille programmer une classe `Cercle` avec comme données-membres :
– un point représentant le `centre`,
– un nombre représentant le `rayon`,
– un nombre représentant la `surface` du cercle.

Permettre l'accès direct à la variable `surface`, c'est s'exposer à ce qu'elle soit modifiée depuis l'extérieur, et cela serait catastrophique puisque l'objet risquerait alors de perdre sa cohérence (la surface dépend en fait du rayon). Il est donc indispensable d'interdire cet accès, ou au moins permettre à l'objet de le contrôler.

(1.3.2) *Données et fonctions-membres d'un objet O seront déclarées publiques si on autorise leur utilisation en dehors de l'objet O, privées si seul l'objet O peut y faire référence.*

Dans la déclaration d'une classe, comment décider de ce qui sera public ou privé ? Une approche simple et sûre consiste à déclarer systématiquement les données-membres *privées* et les fonctions-membres *publiques*. On peut alors autoriser l'accès aux données-membres (pour consultation ou modification) par des fonctions prévues à cet effet, appelées *fonctions d'accès*.

Ainsi, la déclaration de la classe `Cercle` ci-dessus pourrait ressembler à :

classe : `Cercle`	
privé :	*public :*
`centre`	`Fixer_centre`
`rayon`	`Fixer_rayon`
`surface`	`Donner_surface`
	`Tracer`

Dans le cas d'une classe `chronometre` (1.1.4), il suffirait de ne déclarer publiques que les seules fonctions-membres `Afficher` et `Presser_sur_un_bouton` pour que les chronomètres puissent être utilisés normalement, en toute sécurité.

1.4 Stratégie D.D.U

(1.4.1) *En C++, la programmation d'une classe se fait en trois phases : déclaration, définition, utilisation (en abrégé : D.D.U).*

Déclaration : c'est la partie interface de la classe. Elle se fait dans un fichier dont le nom se termine par `.h` Ce fichier se présente de la façon suivante :

```
class Maclasse
{
public:
      déclarations des données et fonctions-membres publiques

private:
      déclarations des données et fonctions-membres privées
};
```

Définition : c'est la partie implémentation de la classe. Elle se fait dans un fichier dont le nom se termine par `.cpp` Ce fichier contient les définitions des fonctions-membres de la classe, c'est-à-dire le code complet de chaque fonction.

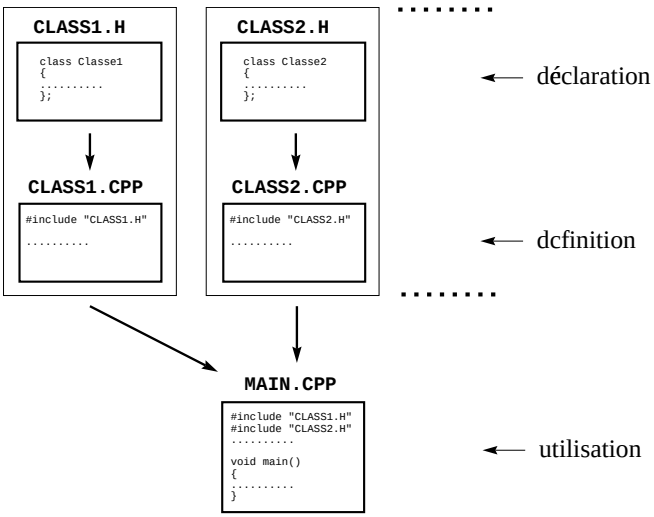
Utilisation : elle se fait dans un fichier dont le nom se termine par `.cpp`

(1.4.2) **Structure d'un programme en C++**

Nos programmes seront généralement composés d'un nombre impair de fichiers :
- pour chaque classe :
  - un fichier `.h` contenant sa déclaration,
  - un fichier `.cpp` contenant sa définition,
- un fichier `.cpp` contenant le traitement principal.

Ce dernier fichier contient la fonction `main`, et c'est par cette fonction que commence l'exécution du programme.

Schématiquement :



(1.4.3) Rappelons que la *directive d'inclusion* `#include` permet d'inclure un fichier de déclarations dans un autre fichier : on écrira `#include <untel.h>` s'il s'agit d'un fichier standard livré avec le compilateur C++, ou `#include "untel.h"` s'il s'agit d'un fichier écrit par nous-mêmes.

1.5 Mise en œuvre

(1.5.1) Nous donnons ici un programme complet afin d'illustrer les principes exposés au paragraphe précédent. Ce programme simule le fonctionnement d'un parcmètre.

Le programme se compose de trois fichiers :

`parcmetr.h` qui contient la déclaration de la classe `Parcmetre`,
`parcmetr.cpp` qui contient la définition de la classe `Parcmetre`,
`simul.cpp` qui contient l'utilisation de la classe `Parcmetre`.

```
// ------------------------ parcmetr.h ------------------------
// ce fichier contient la déclaration de la classe Parcmetre

class Parcmetre
{
public:
    Parcmetre();                      // constructeur de la classe
    void Affiche();                   // affichage du temps de stationnement
    void PrendsPiece(float valeur);   // introduction d'une pièce
private:
    int heures,                       // chiffre des heures...
        minutes;                      // et des minutes
};



// ------------------------ parcmetr.cpp ----------------------
// ce fichier contient la définition de la classe Parcmetre

#include <iostream.h>          // pour les entrées-sorties
#include "parcmetr.h"          // déclaration de la classe Parcmetre

Parcmetre::Parcmetre()         // initialisation d'un nouveau parcmètre
{
    heures = minutes = 0;
}
```

```
void Parcmetre::Affiche()      // affichage du temps de stationnement restant
                               // et du mode d'emploi du parcmètre
{
    cout << "\n\n\tTEMPS DE STATIONNEMENT :";
    cout << heures << " heures  " << minutes << " minutes";
    cout << "\n\n\nMode d'emploi du parcmètre :";
    cout << "\n\tPour mettre une pièce de 10 centimes  : tapez A";
    cout << "\n\tPour mettre une pièce de 20 centimes  : tapez B";
    cout << "\n\tPour mettre une pièce de 50 centimes  : tapez C";
    cout << "\n\tPour mettre une pièce de 1 euro : tapez D";
    cout << "\n\tPour quitter le programme     : tapez Q";
}

void Parcmetre::PrendsPiece(float valeur)     // introduction d'une pièce
{
    minutes += valeur * 10;           // 1 euro = 50 minutes de stationnement
    while (minutes >= 60)
    {
        heures += 1;
        minutes -= 60;
    }
    if (heures >= 3)           // on ne peut dépasser 3 heures
    {
        heures = 3;
        minutes = 0;
    }
}
```

```
// ------------------------ simul.cpp ---------------------
// ce fichier contient l'utilisation de la classe Parcmetre

#include <iostream.h>          // pour les entrées-sorties
#include "parcmetr.h"          // pour la déclaration de la classe Parcmetre

void main()                    // traitement principal
{
    Parcmetre p;               // déclaration d'un parcmètre p
    char choix = 'X';

    while (choix != 'Q')       // boucle principale d'événements
    {
        p.Affiche();
        cout << "\nchoix ? --> ";
        cin >> choix;          // lecture d'une lettre
        switch (choix)         // action correspondante
        {
            case 'A' :
                p.PrendsPiece(1);
                break;
            case 'B' :
                p.PrendsPiece(2);
                break;
            case 'C' :
                p.PrendsPiece(5);
                break;
            case 'D' :
                p.PrendsPiece(10);
        }
    }
}
```

(1.5.2) **Opérateurs . et ::**

Dans une expression, on accède aux données et fonctions-membres d'un objet grâce à la notation pointée : si `mon_objet` est une instance de `Ma_classe`, on écrit `mon_objet.donnee` (à condition que `donnee` figure dans la déclaration de `Ma_classe`, et que l'accès en soit possible : voir (1.3)).

D'autre part, dans la définition d'une fonction-membre, on doit ajouter *<nom de la classe>*`::` devant le nom de la fonction. Par exemple, la définition d'une fonction-membre `truc()` de la classe `Ma_classe` aura la forme suivante :

<*type*> `Ma_classe::truc(`<*déclaration de paramètres formels*>`)`
<*instruction-bloc*>

L'appel se fait avec la notation pointée, par exemple : `mon_obj.truc()` ; en programmation-objet, on dit parfois qu'on envoie le *message* `truc()` à l'objet *destinataire* `mon_obj`.

Exceptions : certaines fonctions-membres sont déclarées sans type de résultat et ont le même nom que celui de la classe : ce sont les constructeurs. Ces constructeurs permettent notamment d'initialiser les objets dès leur déclaration.

(1.5.3) **Réalisation pratique du programme**

Elle se fait en trois étapes :

1) création des fichiers sources `parcmetr.h`, `parcmetr.cpp` et `simul.cpp`.
2) compilation des fichiers `.cpp`, à savoir `parcmetr.cpp` et `simul.cpp`, ce qui crée deux fichiers objets `parcmetr.obj` et `simul.obj` (ces fichiers sont la traduction en langage machine des fichiers `.cpp` correspondants),
3) édition des liens entre les fichiers objets, pour produire finalement un fichier exécutable dont le nom se termine par `.exe`.

Dans l'environnement Visual C++ de Microsoft, les phases 2 et 3 sont automatisées : il suffit de créer les fichiers-sources `.h` et `.cpp`, d'ajouter ces fichiers dans le *projet* et de lancer ensuite la commande `build`.

Remarque.— On peut ajouter directement dans un projet un fichier `.obj` : il n'est pas nécessaire de disposer du fichier source `.cpp` correspondant. On pourra donc travailler avec des classes déjà compilées.