Les classes : rappels

1 Les classes : leur utilité

L'utilisation des classes permet d'assurer certaines propriétés au code : l'encapsulation, la modularité, l'abstraction, l'extensibilité, le polymorphisme. Nous développons ces notions dans les paragraphes suivants. Nous verrons également leur mise en œuvre au cours du module.

L'**encapsulation** signifie que l'on sépare les données internes à une classe du reste du programme. Ceci permet par exemple d'empêcher un utilisateur de mettre des données de la classe dans un état incohérent (par exemple si une variable entière compte le nombre de personnes dans une salle, celle-ci ne doit pas être négative). De la même façon, les fonctions de la classe ne peuvent pas modifier (par inadvertance) des élément extérieurs à celle-ci (sauf si on leur donne en argument). Les classes contiennent en interne toutes les informations dont elles ont besoin pour effectuer leur mission. L'encapsulation diminue donc le risque d'erreurs en séparant les éléments internes d'une classe des éléments externes.

On parle de **modularité** pour désigner le fait qu'on va créer des 'blocs' informatiques contenant les données et les fonctions nécessaires pour effectuer une tache particulière. Ici, bien entendu, chaque 'bloc' correspond à une classe. On peut ensuite faire appel à des fonctions données dans ce 'bloc' pour effectuer la tache demandée. On appellera 'interface' les éléments du 'bloc' accessibles depuis l'extérieur. Un des principaux intérêts de la modularité est qu'on peut remplacer de manière transparente un bloc par un autre, du moment que l'interface reste la même. Ainsi, si on veut utiliser une autre classe pour la même mission, il suffit de lui donner la même interface et de changer la déclaration de la classe et tout le reste du code fonctionne de la même manière. Par exemple, si on résout un système linéaire et que l'on a défini une classe *GradientConjugue* pour calculer la solution de ce système linéaire avec une méthode *solve(Matrice, Vecteur)*, on peut aussi définir une classe *DecompositionLU* avec une autre méthode *solve(Matrice, Vecteur)*. Pour passer d'une méthode à l'autre, il suffira de déclarer le solveur

comme *GradientConjugue* ou comme *DecompositionLU*. Il n'y aura pas besoin de modifications supplémentaires dans le fichier principal.

On parle d'**abstraction** pour désigner le fait qu'un utilisateur peut utiliser une classe à partir de sa documentation en faisant appel à ses fonctions d'interface sans avoir à connaître le détail de son implémentation. Cela permet à l'utilisateur de ne pas se perdre dans les détails et d'utiliser beaucoup plus facilement la classe. C'est particulièrement utile si l'utilisateur n'est pas un expert de la fonctionalité associée à la classe. Par exemple, vous pouvez utiliser une classe permettant de trier une liste par ordre alphabétique sans connaître l'algorithme de tri utilisé (vous avez juste à savoir si cet algorithme est efficace [sous quelles conditions ?] et de quelles données il a besoin).

Par **extensibilité**, on veut dire que les classes permettent de réutiliser des morceaux de code en complétant (ou adaptant) certaines parties de celui-ci. Cette fonctionnalité permet de faire appel à un morceau de code déjà écrit plutôt qu'à le réécrire entièrement. On peut ainsi diminuer la taille d'un programme en ne répétant pas ce qui est déjà écrit, mais de plus, on peut le maintenir plus facilement. Par exemple, si vous vous rendez compte qu'il y a une erreur à un endroit, il est plus facile de corriger cette erreur dans l'unique morceau de code qui la contient que dans toutes les répétitions de ce même morceau. En pratique, on utilisera l'**héritage** de classe pour assurer l'extensibilité du code.

On parle de **polymorphisme** pour désigner le fait qu'on peut vouloir représenter par un même élément informatique une fonction qui peut porter sur différents objets (avec des opérations élémentaires qui peuvent être différentes). Par exemple, on peut définir une fonction *power(A,n)* pour calculer la puissance n d'un élément A. Les opérations à effectuer seront différentes en fonction de si cet élément est un réel, un complexe, une matrice. L'utilisation du polymorphisme peut se faire de différentes manières : on peut par exemple utiliser l'héritage, ou des templates, ou des surcharges de fonction.

2 Les droits d'accès

L'utilisation est très simple : chaque élément de la classe peut être *private*, *public* ou *protected*.

- *private* : accessible uniquement par les membres de la classe.
- *protected* : accessible par les membres de la classe et les membres des classes héritées (si autorisé par l'héritage).
- *public* : accessible par tout le monde.

Les classes et fonctions amies (*friend*) peuvent accéder à tous les éléments, y compris ceux qui sont *private*.

Ces droits d'accès permettent la mise en place de l'encapsulation : on déclare *private* ou *protected* les éléments que l'on ne veut pas voir modifiés par inadvertance. On pourra ensuite par exemple définir une méthode *public* nommée *setX* pour modifier la valeur de l'attribut *X*. Cela permettra de vérifier que la valeur que l'on souhaite donner à *X* ne mène pas à un état incohérent de la classe.

3 Les constructeurs et destructeurs

Un constructeur est une fonction appelée pour générer une instance de la classe. On s'en sert notamment pour allouer l'espace mémoire nécessaire à cette classe et initialiser les valeurs de ses membres. Le destructeur est la fonction appelée automatiquement lorsque l'instance de la classe n'est plus utilisée et donc éliminée. Deux constructeurs et un destructeur sont automatiquement générés lors de la déclaration d'une classe. Les deux constructeurs automatiquement générés sont un constructeur par défaut (sans argument) et un constructeur par copie. Ces constructeurs et destructeur générés automatiquement peuvent (et doivent ! !) être remplacés si on les juge inappropriés.

Un cas particulier où il faut réécrire ces méthodes est le cas où on alloue dynamiquement de la mémoire. Dans ce cas, il faut changer le constructeur par défaut pour que celui-ci alloue la mémoire (sinon on pourra avoir des instances de la classe sans mémoire allouée), il faut changer le destructeur pour que celui-ci libère la mémoire allouée (sinon on va avoir une fuite mémoire dans le code) et il faut changer le constructeur par copie pour ne pas qu'il recopie l'adresse où est allouée la mémoire (sinon les deux instances de la classe ne seront pas indépendantes).

Toutes ces notions seront travaillées lors de la première séance de TP.

4 L'héritage

L'héritage permet à une classe dérivée (ou fille) de reprendre les même caractéristiques qu'une classe de base (ou mère). Ceci permet d'étendre les fonctionnalités de la classe de base à travers la classe dérivée. Les classes dérivées peuvent être utilisées comme classes de base pour définir d'autres classes dérivées. On peut ainsi construire un arbre généalogique de classes. Ceci permet de spécialiser

le code que l'on programme.

Par exemple, on peut définir une première classe *SolveSyst* qui permet de résoudre les systèmes linéaires. Cette classe doit pouvoir être utilisée pour résoudre tous les systèmes linéaires. On pourra par exemple la coder en utilisant une décomposition LU. On peut ensuite définir une classe plus performante mais ne permettant de résoudre que des systèmes linéaires pour lesquels la matrice est symétrique définie positive, on l'appelle par exemple *SolveSystSymPos*. On pourra par exemple coder cela avec une méthode de gradient conjugué ou de Cholesky. Ici, on construira la classe *SolveSystSymPos* comme une classe dérivée de *SolveSyst*. Ainsi toutes les matrices appelées par Cholesky pourront aussi être appelées par LU mais pas l'inverse. De plus, un certain nombre de fonctionnalités déjà présentes dans la classe de base pourront être réutilisées dans la classe dérivée (par exemple des produits scalaires, des produits matrice/vecteur ou tout simplement la décomposition LU). Des tests seront ajoutés lors de l'initialisation de la matrice pour vérifier que celle-ci possède les propriétés attendues. Ceci illustre l'extensibilité permise par l'héritage.

L'héritage permet aussi le polymorphisme (dynamique), ceci signifie que les mêmes opérations pourront être effectuées sur plusieurs objets différents. Par exemple ici, on pourra initialiser une instance *solver* de la classe *SolveSyst* et écrire tout le code sans avoir à se demander si *solver* est en réalité une instance de *SolveSyst* ou de *SolveSystSymPos*. Cela simplifie beaucoup le code utilisé. De plus, on peut faire en sorte que le code détecte lors de son exécution quel type de matrice lui est donnée et il peut s'adapter en choisissant d'appliquer l'algorithme qu'il convient.

Une façon de mettre en place le polymorphisme dynamique est de faire appel à une classe abstraite. Il s'agit d'une classe dont on n'utilisera jamais les instances. Cette classe permet simplement d'être utilisée comme classe de base par d'autres classes qui seront, elles, utilisées. On peut par exemple considérer une classe abstraite *vehicule* avec une méthode *se_deplacer()*. Cette classe ne désigne pas d'objet très précisément et on ne l'utilisera pas directement. On peut ensuite définir deux classes qui héritent de *vehicule* : *voiture* et *velo*. Ces deux classes désignent elles des objets très précis.

L'utilité de cette classe abstraite est que l'on peut maintenant définir une liste de *vehicule* et cette liste peut contenir à la fois des *voiture* et des *velo*. De plus, la méthode *se_deplacer()* sera disponible pour les deux classes bien que son implémentation soit différente entre la voiture (où il faut allumer le moteur, utiliser les différentes pédales, le volant et le levier de vitesses) et le velo (où il faut pédaler).

On appelle **méthode virtuelle** une méthode d'une classe qui peut être redéfinie

dans ses classes dérivées. On utilisera *virtual* pour la définir. On appelle **méthode virtuelle pure** une méthode d'une classe qui n'est pas définie dans cette classe et qui doit être définie dans les classes dérivées. Notons bien le "doit" : si on ne redéfinit pas une méthode virtuelle pure dans une classe dérivée, alors le code ne compile pas. On appelle **classe abstraite** une classe qui a au moins une méthode virtuelle pure. Par exemple *vehicule* est une classe abstraite et *se_deplacer()* est une méthode virtuelle pure dans *vehicule* qui est définie (différemment) dans *voiture* et *velo*.

Droits d'accès pour l'héritage : une classe dérivée peut être définie avec un héritage *public*, *protected* ou *private*. Les droits d'accès aux éléments de la classe de base seront alors déterminés comme étant les moins permissifs entre les droits d'accès dans la classe de base et le droit d'accès de l'héritage. Par exemple, si l'héritage est *public* alors un élément *public* sera *public* et un élément *protected* sera *protected*. Si l'héritage est *private*, les éléments *public* et *protected* seront *private*. Bien entendu, tous les éléments *private* de la classe de base sont indisponibles pour la classe dérivée.

5 Méthodes constantes

On peut indiquer qu'une méthode est **constante** en ajoutant un *const* après les parenthèses contenant les arguments de la méthode. Par exemple

```
int size() const {return n ;}
```

Une telle méthode ne pourra pas modifier les membres de la classe (si on le fait, alors il y a une erreur de compilation).

Si une méthode n'est pas censée modifier les membres de la classe, alors nous vous recommandons de la rendre constante. Ceci permet :

- de s'assurer qu'on ne modifie pas les membres de la classe,
- de donner de l'information au compilateur pour optimiser le code,
- d'indiquer à l'utilisateur que la méthode ne modifie rien,
- à la méthode d'être utilisée par des instances constantes de la classe.