Les principes de la POO

Introduction

Pendant longtemps, la principale préoccupation des programmeurs était de concevoir des applications très courtes capables de tourner rapidement : la mémoire coûtait cher, tout comme le traitement machine. Avec la miniaturisation des ordinateurs, l'augmentation de leurs performances et la chute des prix, les priorités ont changé. A l'heure actuelle, le coût de développement dépasse largement celui d'un ordinateur. L'important est d'écrire des programmes performants, bien construits et faciles à mettre à jour.

Si les ingénieurs en matériel électronique devaient partir d'un tas de sable à chaque fois qu'ils conçoivent un nouveau dispositif, si leur première étape devait toujours consister à extraire le silicium pour fabriquer des circuits intégrés, ils ne progresseraient pas bien vite. Or, un concepteur de matériel construit toujours un système à partir de composants préparés, chacun chargé d'une fonction particulière et fournissant un ensemble de services à travers des interfaces définies. La tâche des concepteurs de matériel est considérablement simplifiée par le travail de leur prédécesseurs.

La réutilisation est aussi une voie vers la création de meilleurs logiciels. Aujourd'hui encore, les développeurs de logiciels en sont toujours à partir d'une certaine forme de sable et à suivre les mêmes étapes que les centaines de programmeurs qui les ont précédés. Le résultat est souvent excellent, mais il pourrait être amélioré. La création de nouvelles applications à partir de composants existants, déjà testés, a toutes chances de produire un code plus fiable. De plus, elle peut se révéler nettement plus rapide et plus économique, ce qui n'est pas moins important."

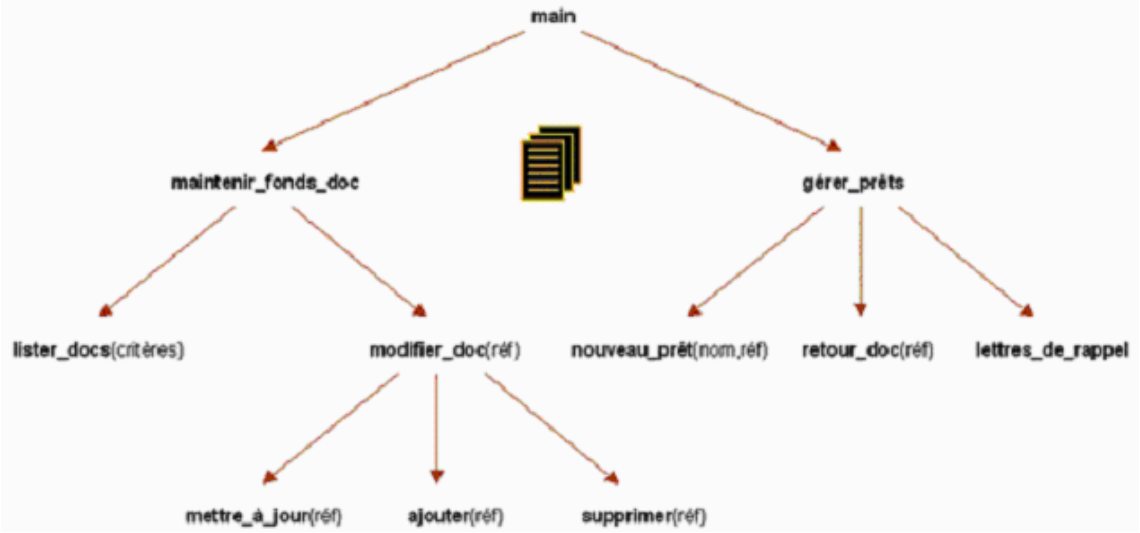
La programmation procédurale

En programmation *procédurale*, les applications sont conçues comme des suites de procédures destinées à traiter des données. La programmation *structurée* a été inventée pour homogénéiser et organiser les interactions entre ces procédures et des volumes importants de données.
Le principe de la programmation structurée est de ***diviser pour mieux régner.*** Ainsi, le programme regroupe de petites unités autonomes de code qui effectuent des opérations simples. La taille de la structure (*module* ou *package*) des procédures facilite la compréhension et la maintenance du programme.

Etude de cas : Gestion d'une bibliothèque

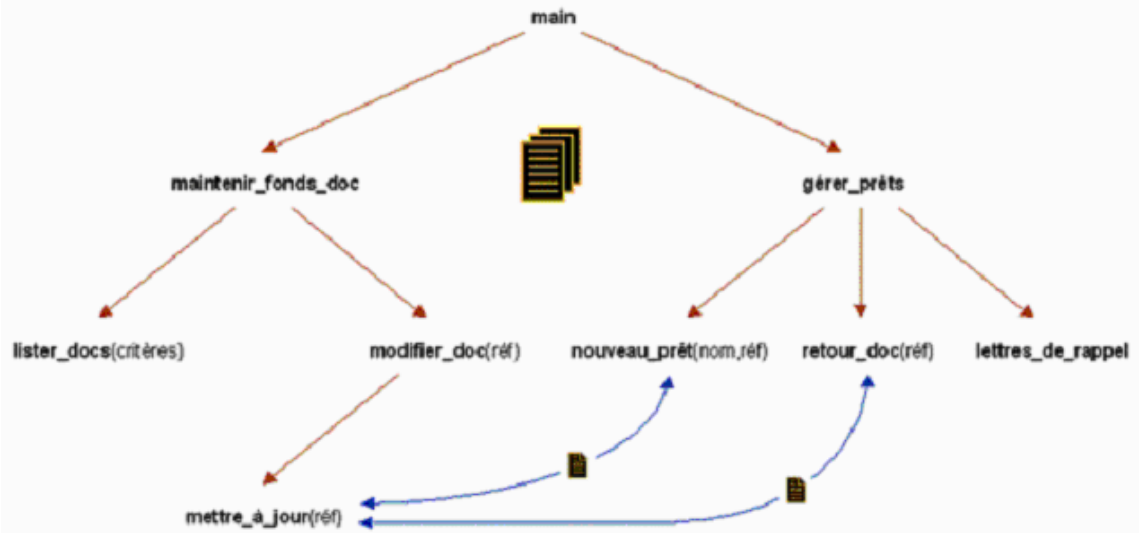
La découpe fonctionnelle d'un problème informatique : une approche intuitive

Exemple de découpe fonctionnelle d'un logiciel dédié à la gestion d'une bibliothèque :



Le logiciel est composé d'une hiérarchie de fonctions, qui ensemble, fournissent les services désirés, ainsi que de données qui représentent les éléments manipulés (livres, etc.). Logique, cohérent et intuitif.

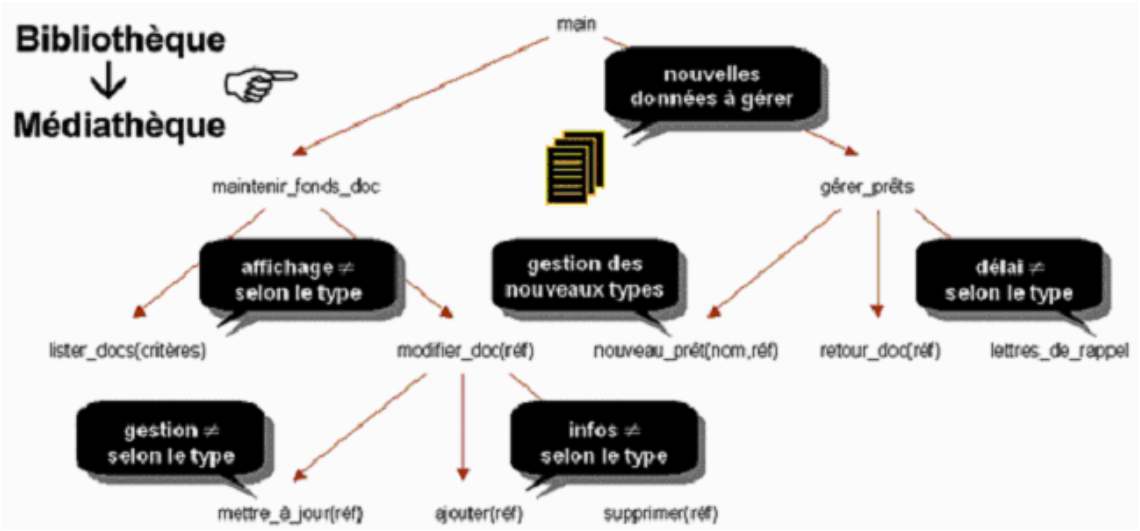
Une découpe fonctionnelle "intelligente" consiste à factoriser certains comportements communs du logiciel. En d'autres termes : pour réaliser une fonction du logiciel, on peut utiliser un ensemble d'autres fonctions, déjà disponibles, pour peu qu'on rende ces dernières un tant soit peu génériques. Génial !



Le revers de la médaille : maintenance complexe en cas d'évolution

Factoriser les comportements n'a malheureusement pas que des avantages. Les fonctions sont devenues interdépendantes : une simple mise à jour du logiciel à un point donné, peut impacter en cascade une multitude d'autres fonctions. On peut minorer cet impact, pour peu qu'on utilise des fonctions plus génériques et des structures de données ouvertes. Mais respecter ces contraintes rend l'écriture du logiciel et sa maintenance plus complexe.

En cas d'évolution majeure du logiciel (passage de la gestion d'une bibliothèque à celle d'une médiathèque par exemple), le scénario est encore pire. Même si la structure générale du logiciel reste valide, la multiplication des points de maintenance, engendrée par le chaînage des fonctions, rend l'adaptation très laborieuse. Le logiciel doit être retouché dans sa globalité :



La séparation des données et des traitements : le piège !

Examinons le problème de l'évolution de code fonctionnel plus en détail...

Faire évoluer une application de gestion de bibliothèque pour gérer une médiathèque, afin de prendre en compte de nouveaux types d'ouvrages (cassettes vidéo, CD-ROM, etc.), nécessite :

- faire évoluer les structures de données qui sont manipulées par les fonctions,
- 'adapter les traitements, qui ne manipulaient à l'origine qu'un seul type de document (des livres).

Il faudra donc modifier toutes les portions de code qui utilisent la base documentaire, pour gérer les données et les actions propres aux différents types de documents.
Il faudra par exemple modifier la fonction qui réalise l'édition des "lettres de rappel" (une lettre de rappel est une mise en demeure, qu'on envoie automatiquement aux personnes qui tardent à rendre un ouvrage emprunté). Si l'on désire que le délai avant rappel varie selon le

type de document emprunté, il faut prévoir une règle de calcul pour chaque type de document.
En fait, c'est la quasi-totalité de l'application qui devra être adaptée, pour gérer les nouvelles données et réaliser les traitements correspondants. Et cela, à chaque fois qu'on décidera de gérer un nouveau type de document !

```
struct Document
{
    char nom_doc[50];
    Type_doc type;
    Bool est_emprunte;
    char emprunteur[50];
    struct tm date_emprunt;
} DOC[MAX_DOCS];

void lettres_de_rappel()
{
    /* ... */
    for (i = 0; i < NB_DOCS; i++)
    {
        if (DOC[i].est_emprunte)
        {
            switch(DOC[i].type)
            {
                case LIVRE:
                delai_avant_rappel = 20;
                break;
                case CASSETTE_VIDEO:
                delai_avant_rappel = 7;
                break;
                case CD_ROM:
                delai_avant_rappel = 5;
                break;
            }
        }
    }
/* ... */
}
```

```
void mettre_a_jour(int ref)
{
    /* ... */
    switch(DOC[ref].type)
    {
        case LIVRE:
        maj_livre(DOC[ref]);
        break;
    case CASSETTE_VIDEO:
        maj_k7(DOC[ref]);
        break;
        case CD_ROM:
        maj_cd(DOC[ref]);
        break;
    }
    /* ... */
}
```

1ère amélioration : rassembler les valeurs qui caractérisent un type, dans le type

Une solution relativement élégante à la multiplication des branches conditionnelles et des redondances dans le code (conséquence logique d'une trop grande ouverture des données), consiste tout simplement à centraliser dans les structures de données, les valeurs qui leurs sont propres :

```
struct Document
{
  char nom_doc[50];
  Type_doc type;
  Bool est_emprunte;
  char emprunteur[50];
```

```
  struct tm date_emprunt;
  int delai_avant_rappel;
} DOC[MAX_DOCS];

void lettres_de_rappel()
{
  /* ... */
  for (i = 0; i < NB_DOCS; i++)
  {
    if (DOC[i].est_emprunte) /* SI LE DOC EST EMPRUNTE */
    {
      /* IMPRIME UNE LETTRE SI SON
      DELAI DE RAPPEL EST DEPASSE */

      if (date() >= (DOC[i].date_emprunt + DOC[i].delai_avant_rappel))
          imprimer_rappel(DOC[i]);
    }
  }
}
```

Quoi de plus logique ? En effet, le "délai avant édition d'une lettre de rappel" est bien une caractéristique propre à tous les ouvrages gérés par notre application.

Mais cette solution n'est pas encore optimale !

2ème amélioration : centraliser les traitements associés à un type, auprès du type

Pourquoi ne pas aussi rassembler dans une même unité physique les types de données et tous les traitements associés ?

Que se passerait-il par exemple si l'on centralisait dans un même fichier, la structure de données qui décrit les documents et la fonction de calcul du délai avant rappel ? Cela nous permettrait de retrouver en un clin d'oeil le bout de code qui est chargé de calculer le délai avant rappel d'un document, puisqu'il se trouve au plus près de la structure de données concernée.

Ainsi, si notre médiathèque devait gérer un nouveau type d'ouvrage, il suffirait de modifier une seule fonction (qu'on sait retrouver instannément), pour assurer la prise en compte de ce nouveau type de document dans le calcul du délai avant rappel. Plus besoin de fouiller partout dans le code...

```
struct Document

{
  char nom_doc[50];
  Type_doc type;
  Bool est_emprunte;
  char emprunteur[50];
  struct tm date_emprunt;
  int delai_avant_rappel;
} DOC[MAX_DOCS];
```

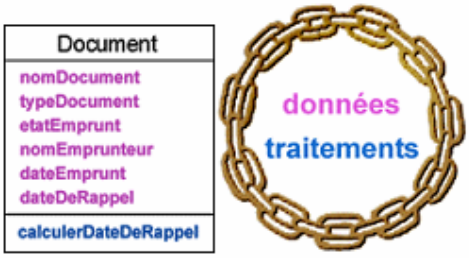
```
int calculer_delai_rappel(Type_doc type)
{
  switch(type)
  {
    case LIVRE:
      return 20;
    case CASSETTE_VIDEO:
      return 7;
    case CD_ROM:
      return 5;
    /* autres "case" bienvenus ici ! */
  }
}
```

Ecrit en ces termes, notre logiciel sera plus facile à maintenir et bien plus lisible. Le stockage et le calcul du délai avant rappel des documents, est désormais assuré par une seule et unique unité physique (quelques lignes de code, rapidement identifiables).

Pour accéder à la caractéristique "délai avant rappel" d'un document, il suffit de récupérer la valeur correspondante parmi les champs qui décrivent le document. Pour assurer la prise en compte d'un nouveau type de document dans le calcul du délai avant rappel, il suffit de modifier une seule fonction, située au même endroit que la structure de données qui décrit les documents.

```
void ajouter_document(int ref)
{
  DOC[ref].est_emprunte = FAUX;
  DOC[ref].nom_doc = saisir_nom();
  DOC[ref].type = saisir_type();
  DOC[ref].delai_avant_rappel = calculer_delai_rappel(DOC[ref].type);
}

void afficher_document(int ref)
{
  printf("Nom du document: %s\n",DOC[ref].nom_doc);

  /* ... */

  printf("Delai avant rappel: %d jours\n",DOC[ref].delai_avant_rappel);

  /* ... */
}
```

En résumé : centraliser les données d'un type et les traitements associés, dans une même unité physique, permet de limiter les points de maintenance dans le code et facilite l'accès à l'information en cas d'évolution du logiciel :



Dualité de base

Fonctions = Opérateurs, actions, processus

Objets = données

En programmation orientée objets, les données et les procédures traitant ces données sont regroupées sous le terme d'**objets** . Il s'agit d'entités autonomes aux propriétés et aux caractéristiques propres.

Qu'est ce que la programmation OO ?

POO : Une première définition

La programmation par objets est la méthode de construction de logiciel qui fonde l'architecture du système sur l'analyse des objets qu'il manipule et non sur le fonctionnement qu'il assure.

« Ne demande pas ce que fait le système mais qu'est ce qu'il manipule . »

La programmation orientée objet permet de concevoir une application sous la forme d'un ensemble d'objets reliés entre eux par des relations : Lorsque que l'on programme avec cette méthode, on se pose plus souvent la question `` **qu'est-ce que je manipule** ? '', que `` **qu'est-ce que je fait** ? ''.

Le L'une Le programme est donc composé d'un assemblage d'objets dont on récupère les propriétés pour les spécialiser à la nouvelle application.

Les ***interfaces graphiques*** se prêtent bien à une conception orientées objets car les objets ne sont pas seulement abstraits mais possèdent une représentation graphique sous la forme d'une fenêtre ou d'un élément graphique.

Langages OO

- C++ (basé sur la langage C)
- Java

Concepts de base de la POO

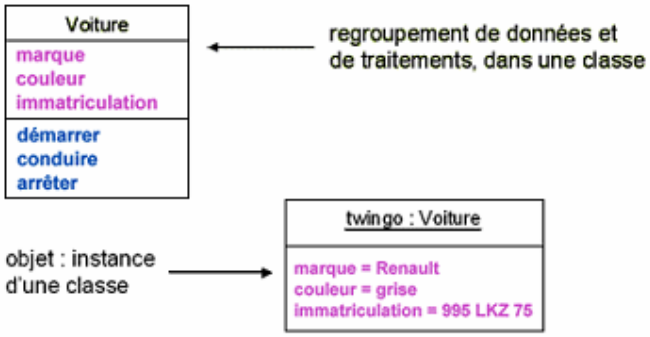
Les classes

Définition

Est une modélisation d'un type du monde réel défini par des aspects:

- Statique : caractéristiques
- Dynamique : opérations

Exemples :



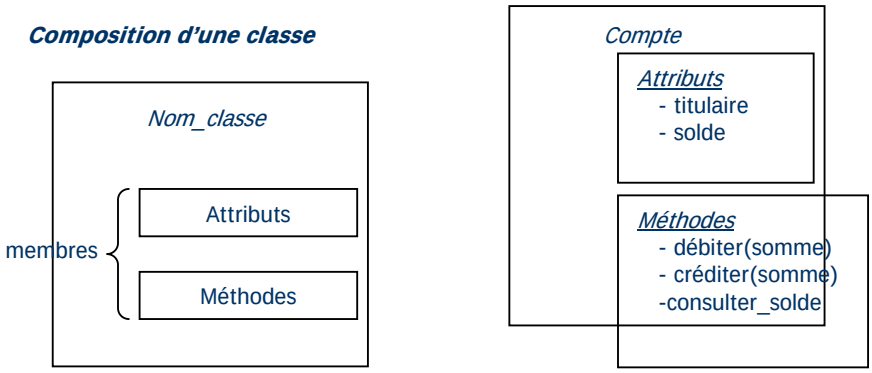
On appelle *classe* la structure d'un objet, c'est-à-dire la déclaration de l'ensemble des entités qui composeront un objet. Un objet est donc "issu" d'une classe, c'est le produit qui sort d'un moule. Un objet est une **instanciation** d'une classe, c'est la raison pour laquelle on pourra parler indifféremment d'*objet* ou d'*instance* (éventuellement d'*occurrence*).

Une classe est composée de deux parties:

- **Les attributs** (parfois appelés *données membres*): il s'agit des données représentant l'état de l'objet
- **Les méthodes** (parfois appelées *fonctions membres*): il s'agit des opérations applicables aux objets

Si on définit la classe *voiture*, les objets *Peugeot 406*, *Renault 18* seront des instanciations de cette classe. Il pourra éventuellement exister plusieurs objets *Peugeot 406*, différenciés par leur numéro de série. Mieux: deux instanciations de classes pourront avoir tous leurs attributs égaux sans pour autant être un seul et même objet. C'est le cas dans le monde

réél, deux T-shirts peuvent être strictement identiques et pourtant ils sont distincts. D'ailleurs en les mélangeant il serait impossible de les distinguer.

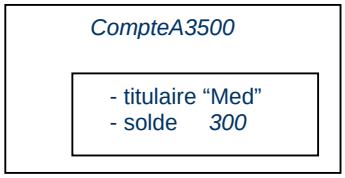


Les objets

Définition

- Un objet est une instance d'une classe
- Il est créé lors de l'exécution

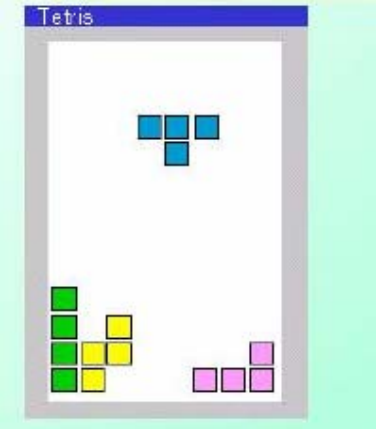
Toute application d'une méthode, accès à un attribut est relative à une instance particulière : instance courante



- CompteA3500. débiter (50) => CompteA3500 .solde = 350
- CompteA3500. titulaire( ) => "Med"
- CompteA3500. consulter_solde( ) => 350

Exemple : le jeu Tetris

Quels sont les concepts/les objets composant le jeu ?
Que doivent savoir faire ces objets ?
Quelles sont leurs propriétés ?



- Objets ?
- Capacité d'action des objets : que font-ils ?
- Attributs des objets : quelles sont leurs propriétés ?

Pièce

- Capacité d'actions :
  - se génère, tombe, tourne,...
  - arrêt en cas de collision
- Attributs
  - orientation
  - position
  - forme

Tableau

- Capacité d'actions
  - se génère
  - efface les lignes pleines
  - teste la fin du jeu
- Attributs
  - Taille
  - Nombre de lignes effacées