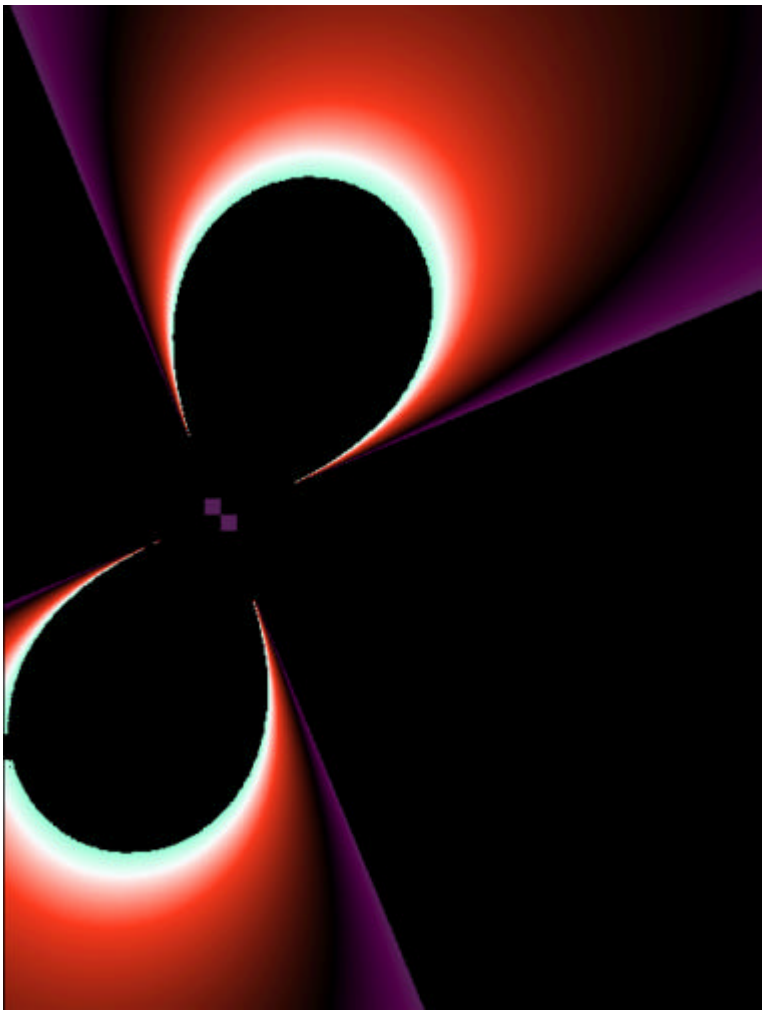
CHAPITRE 3

Qu'est-ce qu'un objet ?



3.1 *Le sens du mot “objet”*

Dans le cadre de la langue française (ainsi que la langue anglaise, d'ailleurs, et beaucoup d'autres), le terme objet cache un peu n'importe quoi. C'est d'ailleurs ce qui fait l'intérêt du terme: on peut sans trop de risques l'utiliser à toutes les sauces, parce que c'est un terme chargé de significations si vagues et si mal définies qu'on ne court pratiquement aucun risque de se trouver détrompé: on ne peut pas avoir tort en parlant d'une chose que personne ne sait définir! Un équivalent populaire du mot objet est "machin".

Dans le cadre informatique pourtant, le terme "objet" est chargé de sens, peut-être trop, d'ailleurs. Puisque nous essayons d'introduire un concept censé, entre autres, de fournir un support à un nouveau paradigme de programmation baptisé "programmation orientée objets", il ne semble pas tout à fait inutile de définir ce que l'on entend par objet. Tenter de définir un terme aussi vague, et volontairement maintenu dans une aura d'obscurité ne peut se faire qu'en émaillant ses dires d'opinions personnelles, et qui de ce fait sont forcément sujettes à critique.

La programmation orientée objet cache plusieurs composantes, parmi lesquelles le langage de programmation ne serait somme tout qu'une partie mineure, si certains détails n'impliquaient pas la nécessité de recourir à un langage spécialisé. Mais il s'agit véritablement de détails techniques: la philosophie transcende le langage: il n'est pas indispensable de disposer de langages typés ou supportant l'héritage pour programmer "orienté objets". En revanche, et quel que soit le langage utilisé, la résolution d'un problème selon des méthodes orientées objet sous-entend une analyse et un effort de conception plus important que l'approche conventionnelle.

3.2 *L'indépendance*

Sous le terme très inapproprié d'objet se cache la notion d'indépendance. Un objet est indépendant de tout l'environnement dans lequel il évolue, sauf des autres objets dont il s'est lui-même déclaré dépendant. En théorie, ceci implique que l'utilisateur d'un objet peut faire tout ce qu'il veut à cet objet: ce dernier réagira toujours de façon appropriée, parceque justement, il n'a pas de liens avec le code qui utilise cet objet. L'objet s'est lui-même défini de telle façon que même une utilisation inappropriée ne peut pas conduire à une faute grossière au niveau de l'objet lui-même. (Par contre, il est évident que l'objet, si bien implémenté soit-il, ne saurait protéger son utilisateur contre lui-même). Si on demande une action inappropriée à un objet, cet objet refusera de faire cette action, parcequ'il ne "comprendra pas" ce que l'on attend de lui. Par contraste, dans un style de programmation classique, il est toujours possible d'appliquer un algorithme quelconque à des données inadéquates, parceque l'algorithme (le code) n'est en aucune façon lié aux quantités qu'il doit traiter. Le fait de contrôler le type de données lors de la compilation ne représente qu'une mesure élémentaire de protection : rien ne permet d'affirmer (sinon de multiples tests impliquant une connaissance profonde de l'implémentation de ce type de données) que la variable du type indiqué est correctement initialisée pour pouvoir être traitée par la procédure en question.

La programmation conventionnelle tend à séparer le code et les données. On a d'une part, un algorithme, et d'autre part des données auxquelles on applique cet algorithme. Cette vision ne s'applique pas uniquement à la génération de code proprement dite, mais également à la phase de conception de projet. Par opposition, notre nouvelle méthode tente de découper le problème global en entités indépendantes, qui englobent les données et les opérations applicables au dites données. Ce conglomérat de code et de données est ce qu'on appelle communément un **objet**, en jargon informatique. Notons que selon le langage de programmation que l'on utilise, cet objet peut faire preuve de caractéristiques très différentes.

3.3 *Messages, méthodes et classes*

Au lieu de passer des données à une procédure, comme on le conçoit dans un environnement de programmation conventionnel, on va émettre des messages à destination d'une instance d'une certaine classe, qui va réagir à ce message en agissant de la manière appropriée sur sa propre implémentation. L'envoi d'un message à un objet implique que l'objet applique une certaine méthode à lui-même. Voici beaucoup de termes nouveaux qu'il convient d'expliciter.

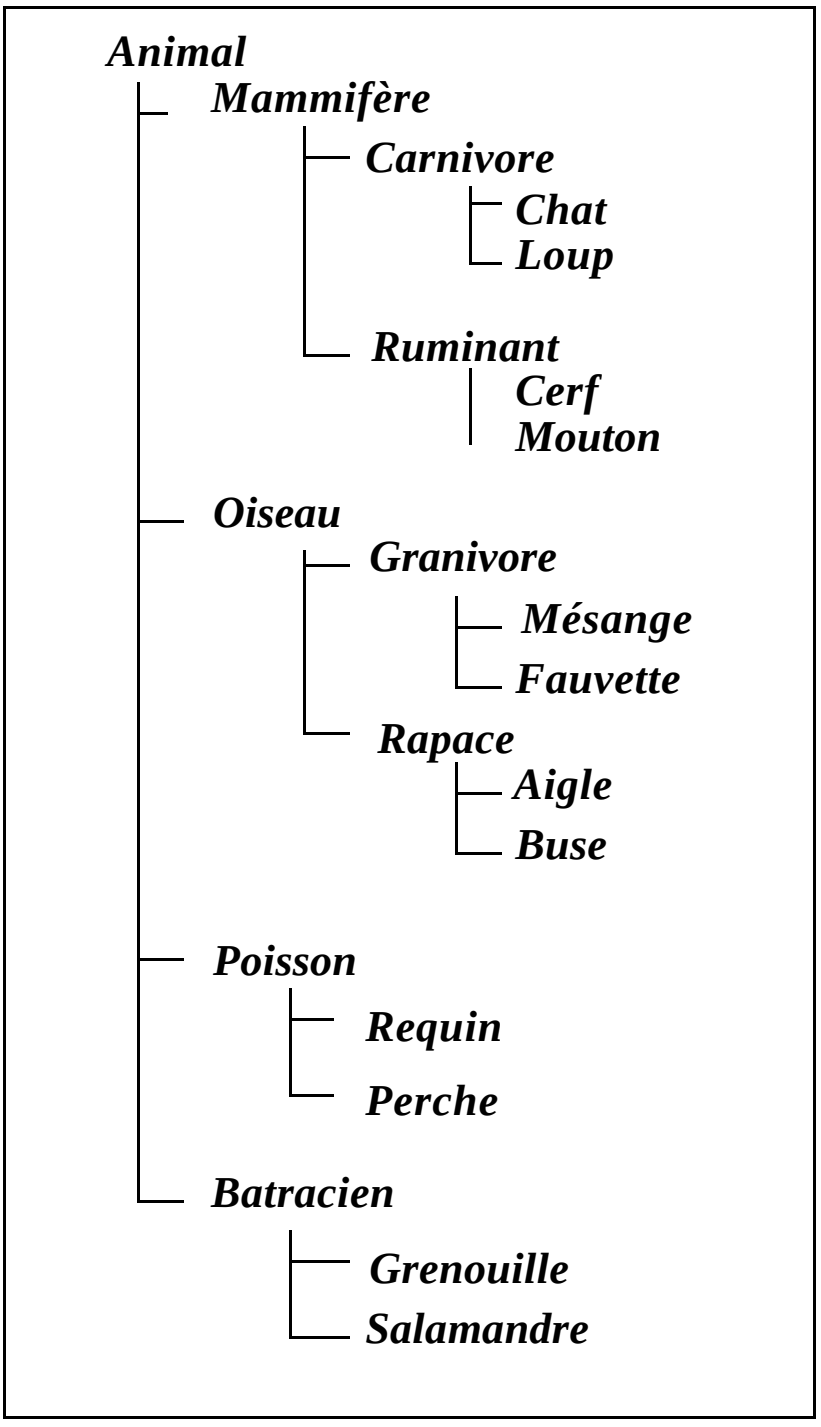
Une classe n'est en fait pas grand'chose d'autre qu'un type de données. La construction PASCAL `RECORD...END` (en C `struct { }`) implémente en principe quelque chose d'assez proche du concept de classe. D'ailleurs, certains jargons de PASCAL, introduits par Borland entre autres, ont défini un dialecte PASCAL "orienté objets" qui utilise la construction RECORD pour implémenter la notion de classe telle qu'elle est définie dans le monde "OO - orienté objets". Si une classe est un type, une instance n'est qu'une incarnation de ce type, comme dans la déclaration:

```
int anInt;
```

int est le type, donc la classe. anInt est la variable de ce type, donc l'instance de cette classe. Ce que OO ajoute à ces notions, c'est qu'une classe est capable de certaines réactions, comme si tout objet de la classe int était capable de réagir à un stimulus quelconque lui demandant de s'afficher lui-même. Le stimulus en question est appelé, en terminologie OO, un message. La manière que la classe considerée utilisera pour réagir à ce message sera une méthode.

Dans la figure ci-dessous, on a représenté une hiérarchie d'animaux. Ces animaux ont tous quelques propriétés communes: par exemple, tous les animaux s'alimentent. Il est donc en principe possible de passer le message MANGER à un animal sans tenir compte du type particulier d'animal dont il s'agit. L'animal comprendra ce message, et réagira en conséquence. Cette réaction peut en revanche être très différente selon l'animal : un chat et une mésange ne mangent pas la même chose, faire manger le chat comme le fait la mésange conduira à plus ou moins brève échéance à la mort du chat par inanition, à moins que la réaction du chat ne soit suffisamment vigoureuse (par exemple, en mangeant la mésange...).

FIGURE 3.1 Hiérarchie d'animaux



Par opposition, si il était nécessaire de coder ce modèle dans un langage traditionnel, comme PASCAL, par exemple, le programmeur serait vraisemblablement amené à définir un segment de code du genre :

```
TYPE
        AnimalType = (Grenouille,
                              Salamandre,
                              Chat,
                              Loup,
                              Cerf,
                              Mouton,
                              Mésange,
                              Perche,
                              Requin);
VAR
        anAnimal : AnimalType;

.....

PROCEDURE manger(thisAnimal : AnimalType);

BEGIN
        CASE              thisAnimal OF

                Grenouille : mangerCommeUneGrenouille(thisAnimal);

                Salamandre : mangerCommeUneSalamandre(thisAnimal);

                Chat : mangerCommeUnChat(thisAnimal);

                Loup : mangerCommeUnLoup(thisAnimal);

                Cerf : mangerCommeUnCerf(thisAnimal);

                Mouton : mangerCommeUnMouton(thisAnimal);

                Mésange : mangerCommeUneMésange(thisAnimal);

                Perche : mangerCommeUnePerche(thisAnimal);

                Requin : mangerCommeUnRequin(thisAnimal);

        ELSE    (* ou OTHERWISE, ou rien (clause non standard)... *)
                writeln('Je ne sais pas ce que mange cet animal ',
                   ord(thisAnimal));
        END;

END;
```

...

Cette manière de faire ne correspond pas à la réalité. Dans la réalité, l'animal sait ce qu'il doit faire lorsqu'il a faim, c'est-à-dire lorsqu'il reçoit de son propre organisme le message "manger". Il n'y a guère que dans les jardins zoologiques que ce modèle est appliqué, parce que l'on trie la nourriture au départ, et l'animal ne reçoit que ce qu'il est susceptible de pouvoir ingurgiter; mais on ne peut pas dire que le jardin zoologique modèle idéalement la vie réelle.

Nonobstant les considérations zoologiques ou écologiques, le segment de code ci-dessus est complexe et difficile à maintenir. Si on ajoute un nouvel animal à la petite ménagerie çi-dessus, il faut impérativement ajouter une référence à ce nouvel animal dans le bloc CASE OF çi-dessus, ainsi que dans la définition du type `AnimalType,` sans quoi le programme nous dira logiquement qu'il ne connaît pas l'animal en question. D'autre part, il est vraisemblable que quelque part ailleurs, on va trouver un CASE OF avec les mêmes étiquettes, mais qui concernera le sommeil ou la soif de ces animaux, cette fois. Là aussi, il faudra intervenir.

Le paradigme orienté objets tend à considérer les choses sous un autre angle : dans la réalité, un animal sait comment faire pour manger, par conséquent le modèle informatique doit également pouvoir exprimer cette connaissance de l'animal. Il doit être possible, dans le modèle informatique, d'écrire quelque chose du genre :

```
send message MANGER to thisAnimal;
```

On s'attend conséquemment à ce que thisAnimal applique la bonne méthode en réponse à ce message, et que le résultat sera l'ingestion de la nourriture appropriée par thisAnimal.

3.3.1 *La séparation des données et des traitements : le piège !*

Examinons le problème de l'évolution de code fonctionnel (ou procédural) plus en détail, au travers d'un autre exemple. Il va s'agir ici de 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 :

- de faire évoluer les structures de données qui sont manipulées par les fonctions,
- d'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 !

L'exemple de code suivant, en C, illustre le problème, même s'il est pour le moins incomplet. On a dispersé l'information "type de document" dans l'ensemble du code, et toute modification du type de document va toucher, en toute logique, l'ensemble du code.

```
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;
     }
     /* ... */
   }
```

3.3.2 *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 !

3.3.3 *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.

3.3.4 *Objet ?*

Les modifications que nous avons apporté à notre logiciel de gestion de médiathèque, nous ont amené à transformer ce qui était à l'origine une structure de données, manipulée par des fonctions, en une entité autonome, qui regroupe un ensemble de propriétés cohérentes et de traitements associés. Une telle entité s'appelle... un objet et constitue le concept fondateur de l'approche du même nom.

Un objet est une entité aux frontières précises qui possède

- une identité (un nom).
- Un ensemble d'attributs caractérise l'état de l'objet.
- Un ensemble d'opérations (méthodes) en définissent le comportement.
- Un objet est une instance de classe (une occurrence d'un type abstrait).
- Une classe est un type de données abstrait, caractérisé par des propriétés (attributs et méthodes) communes à des objets et permettant de créer des objets possédant ces propriétés.

3.4 *Nuance entre message et méthode*

En terminologie "orienté objets", on parle constamment de messages et de méthodes. La différence entre un message et une méthode est subtile, mais tout à fait essentielle. Essayons de comprendre cette différence à travers un exemple en C++ :

```
class     XYCoord
          {
          private :
                    float     x, y;
          public :
                    XYCoord(float xx, float yy) : x(xx), y(yy) {;}
          };

class     GraphicShape
          {
          private :
                    XYCoord   location;
          public :
                    virtual void drawObject()
                    ...
          };

class     CircularShape : public GraphicShape
          {
          private :
                    float              radius;
          public :
                    CircularShape(XYCoord origin, float r) :
                              GraphicShape(origin)
                               { radius = r; }
                    void drawObject()
                    ...
          };
class     RectangularShape : public GraphicShape
          {
          private :
                    float              height, width;
          public :
                    RectangularShape(XYCoord origin, float h, float w)
                              : GraphicShape(origin),
                                        height(h), width(w) {; }
                    virtual void drawObject()
                    ...
          };
class     SquareShape : public RectangularShape
          {
          private :
          public :
                    SquareShape(XYCoord origin, float size) :
                              GraphicShape(origin),
                    height(size), width(size) {; }
                    void drawObject()
                    ...
```

```
        };

int             main()

        {
        GraphicShape*s1,
                                *s2,
                                *s3;
        s1 = new CircularShape(XYCoord(1.0, 1.0), 0.5);
        s2 = new RectangularShape(XYCoord(2.0, 2.0), 1.0, 3.0);
        s3 = new SquareShape(XYCoord(3.0, 3.0), 4.0);
        s1->drawObject();
        s2->drawObject();
        s3->drawObject();
        }
```

Dans notre exemple, s1, s2 et s3 sont tous trois des objets de type GraphicShape. Ils diffèrent les uns des autres uniquement par la manière dont on les a générés (en jargon OO, on dirait "instanciés"). On passe à tous les trois le même message, à savoir drawObject(). De fait, ce même message va générer un appel à trois méthodes différentes, permettant le traçage d'un rectangle, d'un carré et d'un cercle respectivement. C'est là toute la différence entre la méthode et le message: la méthode est le segment de code spécifique à une instance particulière de l'objet qui va permettre de réaliser la fonction demandée par le message (non spécifique) envoyé à l'objet.

3.5 *La notion de polymorphisme*

Notons un fait extrêmement important dans l'exemple ci-dessus, pour simple qu'il soit. Il est impossible au compilateur de déterminer quelle méthode sera appellée pour un objet particulier. On pourrait très bien déplacer les trois instructions `new` dans un autre module de compilation pour interdire toute possibilité d'identification par le compilateur, et cela ne changerait rien. Il est donc nécessaire de pouvoir déterminer quelle méthode est appropriée pour un message donné lors de l'exécution. Ceci implique que le compilateur ou l'éditeur de liens ne résoud pas tous les liens nécessaires, mais doit générer du code pour permettre cette résolution en cours d'exécution. Cette technique est connue sous le nom de **late binding**, et est fondamentale dans le cadre de la programmation orientée objets. Elle permet la définition d'entités logicielles qui réagissent différemment à des messages identiques, selon l'état courant du programme et son histoire : c'est le **polymorphisme**.

Le polymorphisme implique un travail supplémentaire en cours d'exécution. C'est un des principaux reproches que l'on fait à la programmation orientée objets : on prétend qu'elle est moins performante en éxécution que la programmation standard. Cette critique doit être relativisée : il est évident qu'un programme conçu selon le paradigme OO et implémenté dans un langage comme Smalltalk sera moins performant qu'un morceau de code spécifiquement écrit pour la performance en C; mais c'est comparer des pommes et des oranges que de se livrer à ce genre d'exercice.

La programmation orientée objets n'est pas tant une technique de codage qu'une technique de conditionnement, d'"emballage" du code, une technique permettant à un fournisseur de code de proposer des fonctionnalités à un consommateur de code. C'est au niveau de la modification de la relation entre fournisseurs et consommateur de code qu'il faut chercher la principale différence entre la programmation conventionnelle et la programmation orientée objets.

***Test**: Définir un objet*

Donner une définition du mot objet, au sens informatique du terme.

Pourquoi dit-on volontiers qu'un objet est "indépendant" ? Quels sont les avantages de cette notion d'indépendance ?

Quels mécanismes peuvent-ils assurer l'indépendance d'un objet ?

Quels avantages attend-on de la POO relativement à la programmation classique, de style procédural ?

Donner un exemple de polymorphisme

La POO est-elle adaptée à des applications ayant des contraintes temporelles sévères ?