Paradigme objet

Christian Nguyen

Département d'informatique
Université de Toulon

Plan

Introduction

Le paradigme objet est (presque) le dual du paradigme procédural.

	procédural	objet
motivation	traitements	données
question	comment ?	quoi ?
solution	enchaînement de procédures	ensemble d'objets
données	inertes	actives

Il ne s'apprend **pas** à partir d'un langage (la différence entre langages procéduraux et orientés objet ne concerne que quelques mots clés : `class`, `interface`, `extends`, etc.).

Conception

Procédurale : adossée aux algorithmes, séquence de fonctions,

- ⊕ modélisation proche du fonctionnement de la machine,
  calculs de complexité pour estimer la durée des traitements.
- ⊖ modélisation incompréhensible (pour les clients),
  mélange conception et implémentation,
  maintenance et réutilisabilité difficiles.

Objet : adossée aux modèles, ensembles d'objets du domaine.

- ⊕ conception intrinsèquement modulaire,
  implémentation a posteriori,
  maintenance et réutilisabilité facilitées (localité).
- ⊖ représentation parcellaire du projet,
  calculs de complexité difficiles (polymorphisme, etc.).

Concepts

La conception orientée objet s'appuie sur 5 concepts : classe, objet, association, héritage et polymorphisme.

À cela s'ajoute le principe d'encapsulation :

- un objet fournit des services,
- ces services sont des méthodes que d'autres objets peuvent utiliser,
- les attributs ne sont nécessaires qu'en tant qu'implantation d'un service (ils ne sont généralement pas accessibles directement)

Les objets communiquent entre eux pour s'échanger des services : mécanisme de transmission de messages (par appel de méthode).

Classe

C'est un modèle du problème, elle est génératrice d'objets (ou instances de classe) pour lesquels elle définit :

- la liste de ses attributs,
- la liste de ses méthodes,
- la liste des objets avec qui elle peut communiquer.

Attribut : donnée propre à chaque objet de la classe.

Méthode : définit un service des objets.

Attribut

C'est une donnée que possède tout objet de la classe (on parle aussi de propriété ou de donnée membre), qui permet de stocker une valeur unique pour chaque objet.

Important, les attributs ne peuvent être que :

- d'un type primitif : boolean, int, float, ...
- d'un type primitif assimilé : string, date, time, ...
- d'un type collection de types précédents.

Méthode

C'est une fonction ou une procédure classique qui définit un service de l'objet. Elle possède éventuellement des paramètres et une valeur de retour.

La signature d'une méthode est la suivante :

- `type_retour nom(paramètre_formel)`,
- l'appel se fait par l'intermédiaire d'un objet de la classe.

Cas particulier des constructeurs et destructeurs : il n'y a pas de notation spécifique pour les constructeurs et les destructeurs parce qu'ils relèvent de l'implémentation, ce ne sont pas des services.

Objet (ou instance)

C'est la manifestation concrète d'une abstraction, créé à partir d'une classe :

- qui occupe un emplacement mémoire.
- qui dispose des méthodes.
- qui possède des attributs capables de mémoriser les effets des méthodes.

L'accès à un attribut ou à une méthode se fait par la notation pointée : `instance.attribut` ou `instance.operation()`.

À l'intérieur d'une méthode, la référence à l'objet appelant porte un nom spécifique (`self` en Python, `this` en C++ et Java, ...).

Remarque : une instance ne peut pas changer de classe (caractère statique de l'instance).

Plan

Objectifs

Dans le paradigme objet, une fonctionnalité est assurée par la collaboration de plusieurs objets qui s'échangent des services.

Pour qu'un objet puisse utiliser les services d'un autre objet, il faut qu'il en ait connaissance :

- soit en tant que paramètre passé à l'une de ses méthodes,
- soit en tant qu'attribut, dans ce cas c'est une association.

Une association doit avoir un rôle (joué par la classe associée) et une multiplicité (qui précise le nombre, un entier ou un intervalle de valeurs, d'objets liés par association).

Il existe quatre types d'association entre les classes : standard, agrégation, composition et relation de dépendance.

Association standard

Elle décrit une relation dynamique de type “connaît”, c’est la relation par défaut :

- une instance d’une classe connaît une instance d’une autre classe,
- elle est dynamique car chaque instance de la classe ne peut connaître les objets avec lesquels elle est en association qu’au moment de l’exécution.

Exemple :

```
class Professeur:
...
def __init__(self, ppromo):
    self.promo = ppromo
```

Association de type agrégation

Elle décrit une relation dynamique ensembliste non symétrique de type "possède". Une classe joue le rôle d'ensemble et une autre classe le rôle d'élément :

- présence dans la classe agrégat de méthodes pour ajouter ou supprimer des objets agrégés.

Exemple :

```
class Promotion:
...
def __init__(self):
    self.etudiants = []

def add(self, Etudiant pe):
    self.etudiants += [pe]
```

Association de type composition

Elle décrit une relation dynamique de subordination non symétrique de type "est constitué de". La classe composant est une partie intrinsèque et privée de la classe composite :

- l'objet composite a la responsabilité de la création, de la destruction [1] et du stockage de l'objet composé.
  ↳ un objet ne peut être le composant que d'*un* objet composite.

Exemple :

```
class Etudiant:
...
def __init__(self):
    self.compte = Compte()
```

1. généralement resp. dans le constructeur et dans le destructeur

Relation de dépendance

Elle est utilisée pour indiquer une relation lâche entre objets qui n'est pas de nature structurelle (une classe dépend d'une autre sans lien explicite).

Exemples : une méthode d'une classe utilise un paramètre ou définit une variable locale ou possède un type de retour correspondant à une autre classe.

Exemple :

```
from immobilier import Salle

class Professeur:
...
def obtenir_acces(self, psalle: Salle):
    psalle.autoriser(self)
```

Plan

Héritage

C'est une relation statique très forte de type "sorte-de". La relation est statique car les classes sont associées au moment de l'analyse du code.

Elle permet de construire une nouvelle classe à partir d'une classe existante en lui ajoutant de nouveaux attributs et de nouvelles méthodes.

La classe dérivée contient par héritage tous les attributs, toutes les méthodes et toutes les associations (non privés) de la classe parent.

Généralisation et spécialisation

L'héritage permet de répondre à deux objectifs antagonistes.

La généralisation :

- on veut factoriser les éléments communs d'un ensemble de classes (méthodes et associations),
- une super-classe est une abstraction de ses sous-classes.

La spécialisation :

- on veut particulariser la liste des services d'une classe par rapport à une autre,
- une sous-classe est un cas particulier de sa super-classe.

Exemple : carré et rectangle ?

Transtypage (*casting*)

Surclassement (*upcasting*) : consiste à référencer un objet d'une classe dérivée B héritant d'une classe A par une variable de type A (`A a = B()`).

Un objet d'une classe dérivée peut se faire passer pour un objet de sa super-classe. La liste de ses services utilisables s'en trouve réduite à celle de la super-classe.

Déclassement (*downcasting*) : fait de convertir une référence sur une classe A vers une de ses classes dérivées B (plus rare).

Un objet référencé comme un objet de la super-classe est ainsi promu en tant qu'objet de la classe dérivée, on peut ainsi utiliser les services de la classe dérivée.

Cette opération n'est possible que si l'objet est effectivement du type de la classe dérivée ou d'une classe dérivée de celui-ci.

Visibilité des membres

La notion de visibilité porte sur la restriction d'accès aux membres d'une classe que sont attributs, méthodes et associations.

Pour des questions de robustesse et de maintenance, il est important de limiter le plus possible la visibilité des membres :

- la visibilité par défaut doit être privée,
- tous les attributs autres que les constantes et les associations doivent toujours être privés,
- les seules méthodes avec une visibilité publique doivent être les services.

Remarques :

- un objet peut accéder aux attributs privés des autres objets de la même classe,
- les classes dérivées ne peuvent pas diminuer la visibilité d'une méthode redéfinie.

Portée des membres

Les membres d'une classe (attributs, méthodes et associations) peuvent appartenir à une instance ou à une classe (dans ce cas, l'emplacement mémoire des membres est partagé par toutes les instances).

Implications pour les attributs :

- d'instance : valeur distincte pour chaque instance,
- de classe : valeur partagée par toutes les instances. Si une instance modifie la valeur d'un attribut de classe, alors toutes les instances partagent cette modification.

Implication pour les méthodes :

- une méthode de classe ne nécessite pas d'instance pour appeler la méthode, elle s'appelle directement à partir du nom de la classe ; elle ne peut utiliser que des membres de classe.

Portée des noms

La portée des noms est locale aux classes. Elle permet de garder une écriture locale des classes sans se soucier des conflits de nom avec d'autres classes.

Il est donc possible de définir des méthodes de même nom dans des classes sans rapport entre elles.

Le choix de la méthode est fait au moment de l'analyse à partir de la classe de l'objet appelant. La liaison est dite statique (*early binding*)

Surcharge et polymorphisme

La surcharge permet de donner un même nom à des méthodes avec des signatures différentes (*overloading*). Cela permet de définir des procédures adaptées aux paramètres dans une même portée.

Le choix de la méthode est fait au moment de l'analyse à partir de la signature (liste des paramètres effectifs). La liaison est dite statique (*early binding*).

Exemple : la classe `Forme` comporte les méthodes `remplissage(coul : int)` et `remplissage(coul : str)`.

Le polymorphisme permet de donner le même nom à des méthodes de même signature (*overriding*) dans une même hiérarchie.

Le choix de la méthode est fait à l'exécution à partir du type de l'objet appelant. La liaison est dynamique (*late binding*).

Exemple : les classes `Carre` et `Rectangle` comportent la méthode `afficher()`

Classe et méthode abstraites

Une classe abstraite est une classe dont on ne peut pas créer d'instances directement.

Elle est créée pour factoriser des services communs à un ensemble de sous-classes. L'intention est d'obliger à créer des classes dérivées qui auront une représentation concrète dans le domaine.

Une méthode abstraite est une méthode sans code. Le code devra forcément être donné dans chaque classe dérivée concrète. L'intention est d'obliger les classes dérivées à redéfinir la méthode parce qu'elle leur est spécifique.

Conséquences : une méthode abstraite ne peut appartenir qu'à une classe abstraite ; si au moins une méthode est abstraite, la classe doit être abstraite ; par contre, une classe abstraite peut ne contenir que des méthodes concrètes.

Interface

La notion de type est implémentée par une interface, celle-ci définit une liste de services que doit posséder une classe pour être de ce type.

On peut voir une interface comme une classe abstraite où toutes les méthodes sont abstraites. Elle ne définit que la liste des méthodes publiques que la classe doit définir, mais ne donne pas le code de ces méthodes.

Une interface est :

- une structure dont toutes les méthodes sont abstraites,
- une structure qui ne possède pas d'attribut.

Une classe peut implémenter plusieurs interfaces.

Exemple : l'interface `Ordonnable` définit la liste des méthodes `superieur()`, `egal()`, ...