TD1 : Introduction à la programmation objet

Compétences

- Utiliser les concepts de classe et d'objet en POO
- Définir le constructeur et les méthodes d'une classe

La Programmation Orientée Objet : POO

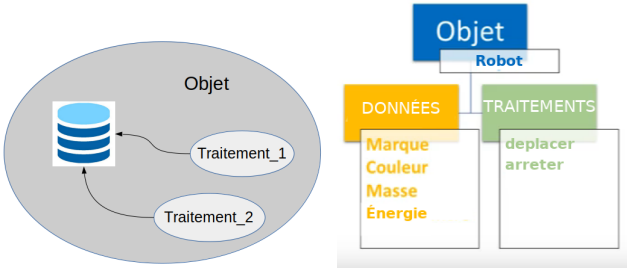
La programmation orientée objet (POO), ou programmation par objet, est un paradigme de programmation informatique. Il consiste en la définition et l'interaction de briques logicielles appelées objets ; un objet représente un concept, une idée ou toute entité du monde physique, comme une voiture, une personne ou un livre. Il possède une structure interne et un comportement, et il sait interagir avec ses pairs. Il s'agit donc de représenter ces objets et leurs relations ; l'interaction entre les objets via leurs relations permet de concevoir et réaliser les fonctionnalités attendues, de mieux résoudre le ou les problèmes. **Dès lors, l'étape de modélisation revêt une importance majeure et nécessaire pour la POO. C'est elle qui permet de transcrire les éléments du réel sous forme d'objets informatiques.**
Un cours de Programmation Orientée Objet par Xavier Blanc.

Un objet c'est quoi ?

Il est très important de bien définir les objets pour le bon déroulement de notre application, de préciser leur rôle dans l'application, leurs traitements et enfin de définir les données dont ils ont besoin avant de se lancer dans la programmation. Cette première étape essentielle est celle de la **conception / modélisation** des objets.

Caractéristiques d'un objet

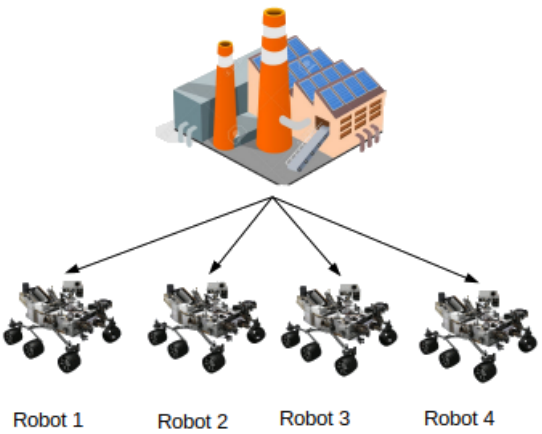
- Son identité
- Ses données
- Le(s) traitement(s) qu'il sait réaliser et qu'il propose aux autres objets



La classe : une usine à objets

Une fois les caractéristiques de notre objet `Robot` définies, on peut fabriquer autant d'objets que nécessaires possédant :

- ses propres données,
- des traitements identiques.



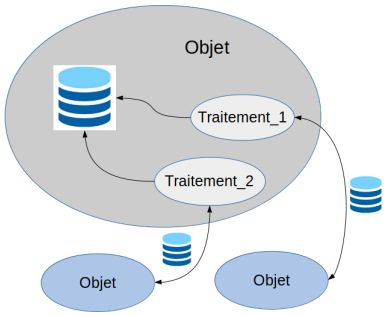
Dans cet exemple chaque objet `Robot` pourra avoir à titre personnel sa couleur, sa marque, sa masse et son énergie mais tous les objets `Robot` seront pourvus des mêmes traitements à savoir : `deplacer` et `arreter`. Cette usine est modélisée par la notion de **classe**.

Communication entre objets

Un objet *A* peut communiquer avec un objet *B* pour lui demander un traitement que *A* ne sait pas faire. Pour envoyer une demande de réalisation de traitement, un objet doit :

L'ESSENTIEL

- Connaître l'ID de l'objet qui va réaliser le traitement.
- Lui envoyer un message avec le nom du traitement et les données qui lui sont nécessaires.
- Recevoir la réponse.



Encapsulation et responsabilité

- **Encapsulation** : L'objet protège ses données afin de préserver son intégrité. Seul l'objet peut modifier ses propres données et seul lui sait comment ses données sont gérées.
- **Responsabilité** : L'objet est responsable des traitements qu'il sait faire et qu'il propose. Ce qui implique qu'il doit impérativement avoir toutes les données nécessaires pour réaliser un traitement qu'il propose. Être responsable ne veut pas dire que l'objet doit travailler seul pour effectuer un traitement, il peut avoir besoin de communiquer avec un ou plusieurs autres objets.

L'utilisateur de l'objet `Robot` n'a accès qu'aux traitements, il ne sait pas comment sont organisées les données utilisées par l'objet `Robot`. On pourrait donc avoir deux objets qui proposent les mêmes traitements mais avec des structures de données différentes.

Les traitements et les données d'un objet doivent former un tout cohérent.

En résumé

Penser une application objet consiste à :

- identifier les objets nécessaires au bon déroulement de l'application,
- préciser les traitements de chaque objet dans l'application,
- définir les données dont ils ont besoin pour réaliser les différents traitements dont ils sont responsables,
- établir les communications,
- faire des objets cohérents.

Remarque : Suivant le contexte les objets de l'application vont être plus ou moins facile à déterminer. Pour illustrer ce propos, prenons deux exemples très simples :

1. Un bateau de pirate qui vient attaquer une île sur laquelle se trouve un fort défendu par des canons.
2. Un jeu de pendu.

Dans le premier exemple, les objets sont presque naturels. Il va en effet y avoir les objets : Bateau, Pirate, Canon... Dans le deuxième exemple les objets sont plus abstraits.

L'application

Une application objet c'est quoi ?

DÉFINITION

Une application objet est un ensemble d'objets qui communiquent pour rendre un service global à son utilisateur.

Deux questions importantes :

1. Comment l'utilisateur interagit avec l'application, et donc avec les objets ?
   → Tous les langages de programmation proposent la possibilité de capter les interactions de l'utilisateur via échange de messages entre un être humain et un objet informatique.
2. Comment sont construits les objets ? Comment l'application démarre-t-elle ?
   → À l'aide d'un objet `Main` qui est un objet un peu particulier permettant de construire tous les objets nécessaires au démarrage de l'application.

Deux règles de bases pour une application objet

CHALLENGE

Les objectifs importants d'une application orientée objet :

- **maximiser la cohérence**, c'est-à-dire avoir autant que possible de petits objets. L'idée est de fabriquer des objets dédiés à des traitements bien spécifiques. **Plus les objets sont petits, plus la cohérence est forte**. Pour comprendre cette notion prenons comme exemple notre robot. Pour l'instant il sait se déplacer d'un point à un autre. Notre objet est cohérent. Si maintenant on lui ajoute la possibilité de modifier sa vitesse, notre objet reste cohérent puisqu'il a besoin d'ajuster celle-ci pour se déplacer et éventuellement faire des économies de carburant. Par contre, si on donne à notre robot la possibilité d'effectuer une analyse de roche, alors nous avons un objet robot qui sait faire deux choses complètement indépendantes et l'objet devient alors incohérent. Les analyses doivent être déléguées à un autre objet.
- **minimiser le couplage**, c'est-à-dire avoir peu (voire pas du tout) de communication entre les objets. Moins il y a de communication, plus le couplage est faible. La question à se poser est : **est-ce qu'un objet peut-être trop ou mal couplé ?** Ce qui est sûr c'est qu'un objet qui ne possède aucun couplage est un objet isolé qui ne présente pas (ou peu) d'intérêt pour notre application. À l'opposé, un objet trop couplé est un objet qui sera difficile à tester car possédant trop de dépendances. Pour finir, on pourra également regarder comment les objets sont couplés. On peut par exemple imaginer un couplage sous forme de cycle entre plusieurs objets. Et bien sûr plus le cycle est grand, moins le couplage est bon.

  Pour approfondir : Cohérence et couplage

Comment construire une classe en Python ?

Le constructeur d'une classe

DÉFINITION

Le constructeur de la classe est la méthode qui sera appelée lors de la création de l'objet

En Python :

- Commence par `def __init__(self):`
- Peut avoir des paramètres transmis lors de la création de l'objet.
- Permet d'initialiser des variables liées à un objet.

Les traitements

Chaque traitement d'un objet possède un nom. Ces traitement sont aussi appelés **`méthodes`**. Une méthode se définit comme une fonction.

En Python :

- Commence par `def nomMethode(self):`
- Peut avoir des paramètres transmis lors de l'utilisation de la méthode.
- Réalise un traitement.

- Peut retourner le résultat du traitement avec le mot clé `return`.

```python
class Humain:

    def __init__(self, genre:str):

        self.__genre = genre
        self.__vie = 100

    # Les méthodes de ma classe
    def manger(self) -> None:

        if self.__vie >= 95:
            self.__vie = 100
        else:
            self.__vie += 5

    def getVie(self) -> int:

        return self.__vie
```

La notion d'héritage

L'héritage en programmation orientée objet (POO) est un concept fondamental qui permet à une classe (appelée la classe dérivée ou sous-classe ou classe fille) d'hériter des propriétés (attributs et méthodes) d'une autre classe (appelée la classe de base ou super-classe ou classe mère). Cela signifie que la classe dérivée peut utiliser et étendre les fonctionnalités de la classe mère, facilitant ainsi la réutilisation du code.

En d'autres termes, l'héritage permet de modéliser une relation **est un** entre les classes. Par exemple, si vous avez une classe Habitant dans un jeu de simulation avec des propriétés telles que nom et âge, vous pouvez créer des classes dérivées telles que Artisan ou Fermier qui hériteront des propriétés de la classe mère Habitant, tout en ayant également leurs propres fonctionnalités spécifiques.

Exemple

Classe mère : Habitant

- Propriétés : nom, âge
- Méthode : afficher_info

Classe dérivée : Fermier (hérite de Habitant)

- Propriété spécifique : culture
- Méthode : cultiver

Classe dérivée : Artisan (hérite de Habitant)

- Propriété spécifique : artisanat
- Méthode : fabriquer

Implémentation

L'implémentation et l'utilisation en Python donne :

```python
class Habitant:
    def __init__(self, nom, age):
        self.nom = nom
        self.age = age

    def afficher_info(self):
        print(f"Nom: {self.nom}, Âge: {self.age}")
```

```python
class Fermier(Habitant):
    def __init__(self, nom, age, culture):
        super().__init__(nom, age)
        self.culture = culture

    def cultiver(self):
        # le code

class Artisan(Habitant):
    def __init__(self, nom, age, artisanat):
        super().__init__(nom, age)
        self.artisanat = artisanat

    def fabriquer(self):
        # le code
```

Avec ces classes, vous pouvez maintenant créer des instances d'Habitant, Fermier et Artisan, et appeler leurs méthodes respectives. Voici un exemple d'utilisation :

```python
# Création d'instances
habitant = Habitant("Alice", 30)
fermier = Fermier("Bob", 40, "Céréales")
artisan = Artisan("Charlie", 25, "Travail du bois")

# Appel des méthodes
habitant.afficher_info()
fermier.afficher_info()
fermier.cultiver()
artisan.afficher_info()
artisan.fabriquer()
```

On remarquera que l'objet `fermier` peut utiliser la méthode `afficher_info` de la classe Habitant dont il hérite. C'est une fonctionnalité à part entière de l'objet `fermier`. L'héritage favorise la hiérarchie, la modularité et la réutilisation du code, car les modifications apportées à la classe mère se répercutent automatiquement sur les classes dérivées. Cela contribue à la création de structures de programme plus flexibles et évolutives dans le cadre de la POO.