M2103 – Structures de Données

Semestre 2 2017-2018

Cours 1

Introduction – Implémentation des SDD

Enseignant

- Dr Mohammed Belkhatir
- Bureau 2.12
- E-mail : mohammed.belkhatir@univ-lyon1.fr

Modalités

- Cours : notes + transparents (à connaître sur le bout des doigts)
- Exercices : sur place ou à emporter
- Page du cours

  https://universite-lyon.academia.edu/MohammedBelkhatir/M2103-SDD

- Evaluation
  - Interrogation(s) surprise
  - Examen

Programme

- Introduction et Implémentation des SDD
  - Présentation de la notion de SDD
  - Implémentation avec un langage de POO
- Allocation Dynamique et Listes Chaînées
- Piles
- Files
- Dictionnaires
- Tables de Hachage

Importance des SDD

- Dans ce cours, on s'intéresse à :
  - La description de structures de données réutilisables dans différents contextes de résolution de problèmes
  - Aux traitements élémentaires permettant leur manipulation

- Essentiels pour la résolution de problèmes
  - Abstraction du problème
  - Algorithme(s)
  - **Structures de données**
  - Implémentation

Abstraction du Problème

- Identification du problème abstrait sans les détails non nécessaires
- Décision réalisée sur les attributs essentiels
- Exemple
  - Pour calculer la factorielle d'un entier positif n, je dois calculer…

Algorithmes

- Suite d'instructions pour résoudre le problème
- Chaque étape doit être *implémentable*.
- Correction
- Terminaison
- Exemple

    factorielle (n) :

        = 1, si n = 0

        = n * factorielle (n-1), sinon

Structures de Données

- Structures dont le but est de stocker des données manipulables par les algorithmes
- Indépendantes des langages de programmation
- Le but ‘ultime’ est de spécifier la nature des opérations sur les structures, pas la manière dont elles fonctionnent

Exemples de Structures de Données

- Liste chaînée
  - Allocation dynamique, insertion facilitée, accès séquentiel lent
- Pile
  - LIFO, récursivité, peut être implémentée par un tableau
- File
  - FIFO, peut aussi être implémentée par un tableau
- Dictionnaires etc.

Opérations des structures de données

- Par position
  - Insertion de données à la position *i* de la structure
  - Effacement de données à la position *i* de la structure
  - Questions concernant les données à la position *i* de la structure
- Par valeur
  - Insertion de données avec valeur x dans la structure
  - Effacement de données avec valeur x dans la structure
  - Questions concernant les données avec valeur x

Implémentation des SDD - POO

- Séparation entre
  - Interface : partie visible correspondant aux opérations que la structure de données peut effectuée.
  - Implémentation : partie cachée qui contient les entités internes permettant de réaliser les opérations de la SDD.
- Données accessibles seulement au travers de l'interface définie (ou des opérations proposées)
- Nous en présenterons ici deux notions :
  - Encapsulation
  - Généricité

Objet

- C’est une « partie de programme » composée d’**attributs** et de **méthodes**.
- **Attributs**
  - Variables locales de l'objet : valeurs caractérisant l'objet
- **Méthodes**
  - Fonctions ou procédures que possède l'objet
  - Effectuent un traitement sur **les attributs de cet objet**

Exemple d'objet : l’objet Personne

- **Attributs**
  - nom, adresse et date de naissance d’une personne

- **Méthodes** :
  - Initialisation des attributs à partir de valeurs données en paramètres
  - Affichage des attributs de la personne à l'écran
  - Changement de son adresse
  - Comparaison du nom de la personne avec un nom donné

Classe d'objets

- **Type d'objets** ayant tous :
  - les mêmes attributs
  - les mêmes méthodes.
- Deux objets de même type :
  - différent seulement par les valeurs de leurs attributs,
  - ont les mêmes méthodes.
- Une classe a un identificateur (nom) et s'utilise comme un type ordinaire.

- **Exemple de classe d'objets**
  - Classe des objets représentant des personnes.
  - Deux personnes ont mêmes attributs et mêmes méthodes,
  - Elles diffèrent seulement par la valeur des attributs *nom* et/ou *adresse*

Instances et Encapsulation

- L'utilisateur de l'objet ne devrait pas avoir a connaître sa structure interne, mais seulement la spécification de son comportement, défini par ses méthodes

- Les instances de classes contiennent également des données
  - Accessibles uniquement aux méthodes de l'objet, c'est pourquoi on les nomme attributs privés
  - En Python, on les caractérise en commençant leur nom par '_'

Envoi de message à un objet

- Syntaxe pointée pour accéder aux méthodes
- `p.changeAdresse("15 rue des fleurs")`

  est un **envoi d'un message à l'objet p**
- Ce message demande à l'objet p d'utiliser sa méthode `changeAdresse` pour **modifier son adresse**
- Un envoi de message à un objet est composé :
  - de **l'objet** auquel est envoyé le message (**`p`**)
  - d'un point (•)
  - du **nom de la méthode** à utiliser (`changeAdresse`)
  - et de **valeurs pour les paramètres** `("15 rue des fleurs")`

Paramètre implicite d'une méthode

- `p.changeAdresse("15 rue des fleurs")`

  range la chaîne "15 rue des fleurs" dans l'attribut adresse de l'objet **p**.
- `changeAdresse` a donc réellement **2 paramètres** :
  - l'objet p auquel est envoyé le message :
    - paramètre particulier
    - spécifié avec une notation particulière (**p.changeAdresse**)
  - "15 rue des fleurs", paramètre classique.
- L'objet qui reçoit le message n'apparaît pas dans la liste des paramètres de la méthode : c'est un **paramètre implicite** de la méthode

Déclaration d'une classe (Python)

- Contenue dans un bloc avec 'class', suivi du nom de la classe puis ':'
- Bloc subdivisé en deux parties : Interface et Implémentation
- Interface
  - Contient la spécification des méthodes
  - C'est la partie publique de la classe (i.e. ce qu'on a besoin de connaître pour écrire un programme utilisant la classe)
- Implémentation
  - Contient la déclaration des attributs privés et l'implémentation des méthodes (i.e. nécessaire à l'ordinateur pour l'exécution de ce programme)
  - Peut également contenir des déclarations de types structurés
  - Peut également contenir la déclaration de procédures et fonctions additionnelles dites *méthodes privées* (elles commenceront par '_')

Accès à l'objet qui reçoit le message - **self**

- La déclaration d'une méthode est similaire à la déclaration d'une procédure ou fonction comme vu au S1, à l'exception du paramètre implicite représentant l'objet dont on appelle la méthode : **self**
- **self** est toujours le premier paramètre de la méthode
- Il est indiqué sans type
- Il doit apparaître comme paramètre d'entrée, de sortie ou d'entrée-sortie, afin d'indiquer le rôle de la méthode sur l'objet auquel on l'applique :
  - Consultation (entrée)
  - Modification (entrée/sortie)
  - Initialisation (sortie)
- Dans une méthode de la classe `Personne`, les attributs de l'objet qui reçoit le message sont donc désignés par :

  `self._nom` et `self._adresse`

Méthodes d'initialisation

- En Python, l'initialiseur est unique, et toujours nommé __init__
- Il peut avoir des paramètres d'entrée, mais self est le seul paramètre de sortie
- Il a la responsabilité d'initialiser les valeurs des attributs privés de l'objet
- Il est appelé implicitement à la création de l'objet
- Pour créer un objet, on écrit : **p = Personne()**

Classe FeuTricolore : Interface

```
class FeuTricolore:

    def __init__(self):
        """
        :sortie self:
        :post-cond: le feu est au vert
        """
    def etat(self):
        """
        :entrée self:
        :sortie e: str
        :post-cond: e est l'état courant du feu, parmi "vert", "orange" ou "rouge"
        """
    def change_etat(self):
        """
        :entrée-sortie self:
        :post-cond: passe à l'état suivant, selon le cycle "vert" -> "orange" -> "rouge" -> "vert"
        """
```

Classe FeuTricolore : Implémentation

```
def __init__(self):
  #### attribut privé
  self._num_etat = 0
  # parmi 0 (vert), 1 (orange), 2 (rouge)

def etat(self):
  if self._num_etat == 0:
     e = "vert"
  elif self._num_etat == 1:
     e = "orange"
  else:
     e = "rouge"
  return e
```

```
def change_etat(self):
  self._num_etat = self._num_etat_suivant()

#### méthode privée
def _num_etat_suivant(self):
  """
  :entrée self:
  :sortie s: int
  :post-cond: s est le num. du prochain état
  """
  s = (self._num_etat + 1) % 3
  return s
```

Classe Vecteur : Interface

```
def __init__(self):
  """
  :sortie self:
  :post-cond: coordonnées du vecteur intialisées à 0
  """
def base(self, v):
  """
  :entrée-sortie self:
  :entrée-sortie v: Vecteur
  :post-cond: v forme une base de l'espace vectoriel
      avec le vecteur courant (vecteurs non nuls et
      colinéaires)
  """
def egaux(self, v):
  """
  :entrée self:
  :entrée v: Vecteur
  :sortie eq: bool
  :post-cond: eq est True si v et le vecteur courant
      sont égaux
  """
def nul(self):
  """
  :entrée self:
  :sortie n: bool
  :post-cond: n est True si le vecteur courant est nul
  """
def colineaires(self, v):
  """
  :entrée self:
  :entrée v: Vecteur
  :sortie co: bool
  :post-cond: co est True si v et le vecteur courant sont colinéaires
  """
def addition(self, v):
  """
  :entrée v: Vecteur
  :entrée-sortie self:
  :post-cond: v est additionné au vecteur courant
  """
def multiplication(self, k):
  """
  :entrée k: float
  :entrée-sortie self:
  :post-cond: le vecteur courant est multiplié par k
  ""
def affiche(self):
  """
  :entrée self:
  :post-cond: affichage des coordonnées du vecteur
  """
```

Classe Vecteur : Implémentation

```
def __init__(self):
  self._x = 0
  self._y = 0

def base(self, v):
  self._x = 0
  self._y = 1
  v._x = 1
  v._y = 0

def egaux(self, v):
  eq = (self._x == v._x and self._y ==
     v._y)
  return eq

def nul(self):
  n = (self._x == 0 and self._y == 0)
  return n
```

```
def colinéaires(self, v):
  co = (not self.nul() and not v.nul() and
  (self._x * v._y - v._x * self._y ) == 0)
  return co

def addition(self, v):
  self._x = self._x + v._x
  self._y = self._y + v._y

def multiplication(self, k):
  self._x = k * self._x
  self._y = k* self._y

def affiche(self):
  print ("les coordonnées sont %f et
  %f " % (self._x,self._y))
```

Classes Génériques

- Certaines classes peuvent être déclinées de différentes manières
  - ex. les structures appelées à stocker plusieurs valeurs d'un même type comme les piles, files...
  - implémentation ne dépend pas du type de valeur stockée
- Idée : on souhaiterait donc pouvoir écrire ces classes une seule fois, et les réutiliser quel que soit le type de valeur qu'on souhaite y stocker.

Classes Génériques avec Python

- En Python et dans les langages faiblement typés, on se contentera en général de ne pas contraindre le type des éléments manipulés par la classe, ou d'utiliser le type le plus abstrait (**object** en Python).
- On utilisera ensuite les pré/post-conditions pour imposer le type des éléments d'un objet donné.
  - Par exemple, une fonction prenant en paramètre d'entrée une pile pourra imposer dans ses pré-conditions que cette pile ne contienne que des entiers
  - Mais le respect de cette pré-condition restera de la responsabilité du programmeur!