Chapitre 2 - Structures de données : Interface et implémentation

I - Définitions

Définition 1

Structure de données : Organisation d'une collection de données en vue de leur exploitation efficace (accès, modification, ...). Elle regroupe des données à gérer et un ensemble d'opérations qu'on peut leur appliquer.

Définition 2

Interface (ou Type abstrait de données) : Vue « logique » de la structure de données. Elle spécifie la nature des données ainsi que l'ensemble des opérations permises sur la structure.

Définition 3

Implémentation : Vue « physique » de la structure de données. Il s'agit de la programmation effective des opérations définies dans l'interface, en utilisant des types de données existants.

Remarque importante

L'interface est la partie visible pour qui veut utiliser ce type de données. Elle précise comment utiliser la structure de données sans se préoccuper de la façon dont les choses ont été programmées (son implémentation).

II - Exemple : la structure `Rationnel`

1) Interface de la structure de données

On aimerait définir une structure de données appelée `Rationnel` correspondant à l'ensemble des nombres rationnels (noté $\mathbb{Q}$).
Voici les opérations que l'on souhaite effectuer sur les rationnels :

- Créer un rationnel
- Accéder au numérateur et au dénominateur d'un rationnel
- Ajouter, soustraire, multiplier, diviser deux rationnels
- Vérifier si deux rationnels sont égaux ou non

On spécifie l'ensemble des opérations souhaitées en proposant l'interface suivante :

- `creerRationnel(n, d)` : crée un élément de type Rationnel à partir de deux entiers `n` (numérateur) et `d` (dénominateur). Précondition : $d \neq 0$.
- `numerateur(r)` : accès au numérateur du rationnel `r` (renvoie un entier)
- `denominateur(r)` : accès au dénominateur du rationnel `r` (renvoie un entier non nul)
- `ajouter(r1, r2)` : renvoie un nouveau rationnel correspondant à la somme des rationnels `r1` et `r2`
- `soustraire(r1, r2)` : renvoie un nouveau rationnel correspondant à la différence des rationnels `r1` et `r2`.
- `multiplier(r1, r2)` : renvoie un nouveau rationnel correspondant au produit des rationnels `r1` et `r2`
- `egal(r1, r2)` : renvoie `Vrai` si les deux rationnels `r1` et `r2` sont égaux, `Faux` sinon.

On ajoute à cela une opération permettant d'afficher un rationnel sous la forme d'une chaîne de caractères :

- `afficher(r)` : affiche le rationnel `r` sous la forme d'une chaîne de caractères `'n/d'` où `n` et `d` sont respectivement le numérateur et le dénominateur de `r`.

2) Exemple d'utilisation de la structure

L'interface apporte toutes les informations nécessaires pour utiliser le type de données. Ainsi, le programmeur qui l'utilise n'a pas à se soucier de la façon dont les données sont représentées ni de la manière dont les opérations sont programmées. L'interface (uniquement) lui permet d'écrire toutes les instructions qu'il souhaite et obtenir des résultats corrects. Par exemple, il sait qu'il peut écrire le programme suivant pour manipuler le type `Rationnel` (écrit ici en Python mais on pourrait le faire dans un autre langage)

```
r1 = creerRationnel(1, 2) # r2 represente le rationnel 1/2
den = denominateur(r1)  # den vaut donc 2
r2 = creerRationnel(1, 3*den)  # r2 represente le rationnel 1/6
r = ajouter(r1, r2)  # r est le resultat de 1/2 + 1/6
egal(r, creerRationnel(2, 3)) # doit renvoyer True puisque 1/2 + 1/6 = 2/3
```

3) Implémentations

Pour implémenter une structure de données, c'est-à-dire programmer les opérations sur la structure, il est nécessaire d'utiliser les structures de données déjà définies dans le langage utilisé. En Python, on peut avantageusement utiliser les types `int`, `float`, `boolean` ainsi que les tuples (avec le type `tuple`), les tableaux (avec le type `list`), les dictionnaires (avec le type `dict`), les ensembles (avec le type `set`).

Nous allons proposer deux implémentations : l'une avec un tuple (on aurait pu utiliser un tableau de manière identique) et l'autre avec un dictionnaire.

Une implémentation possible

On peut par exemple implémenter (= programmer concrètement) le type abstrait `Rationnel` en utilisant des couples (le type `tuple` de Python). Voici ce que pourrait alors être l'implémentation de certaines des opérations du type `Rationnel`.

```
# IMPLEMENTATION AVEC UN COUPLE

def creerRationnel(n, d):
    """Entier x Entier --> Rationnel"""
    return (n, d)

def denominateur(r):
    """Rationnel --> Entier"""
    return r[1]

def ajouter(r1, r2):
    """Rationnel x Rationnel --> Rationnel"""
    # calculs du numerateur et du denominateurs en procedant a une reduction au
       meme denominateur
    num = r1[0] * r2[1] + r2[0] * r1[1]
    den = r1[1] * r2[1]
    # simplification du rationnel en divisant le numerateur et le denominateur par
       leur pgcd
    d = pgcd(num, den)
    return (num // d, den // d)

def egal(r1, r2):
    """Rationnel x Rationnel --> Booleen"""
    return r1 == r2

def afficher(r):
    """Rationnel --> str"""
    print(str(r[0]) + "/" + str(r[1]))

# Une fonction ne faisant pas partie de l'interface de la structure de donnees \
# mais utilisee dans la fonction ajouter
def pgcd(a, b):
    """Renvoie le pgcd des entiers positifs a et b"""
    return a if b == 0 else pgcd (b,a) if b > a else pgcd (a-b, b)
```

On peut alors vérifier, même si on le savait déjà grâce à l'interface, que les instructions précédentes donnent des résultats corrects (on a pris le soin d'afficher certains résultats pour illustrer).

```
r1 = creerRationnel(1, 2)
afficher(r1)                    # pour verifier
den = denominateur(r1)
print(den)                      # pour verifier
r2 = creerRationnel(1, 3*den)
afficher(r2)                    # pour verifier
r = ajouter(r1, r2)
afficher(r)                     # pour verifier
egal(r, creerRationnel(2, 3))
```

```
#OUTPUT
1/2
2
1/6
2/3
True
```

Une autre implémentation possible

Imaginons que le programmeur qui a implémenté le type abstrait `Rationnel` ait fait des choix différents :

- il a utilisé des dictionnaires pour représenter les rationnels
- il a choisi de ne pas simplifier les fractions au fur et à mesure des calculs, ce qui implique une autre écriture du test d'égalité

```
# IMPLEMENTATION AVEC UN DICTIONNAIRE ET PAS DE SIMPLIFICATIONS INTERMEDIAIRES

def creerRationnel(n, d):
    """Entier x Entier --> Rationnel"""
    return {"num": n, "den": d}

def denominateur(r):
    """Rationnel --> Entier"""
    return r["den"]

def ajouter(r1, r2):
    """Rationnel x Rationnel --> Rationnel"""
    # calculs du numerateur et du denominateurs en procedant a une reduction au
       meme denominateur
    num = r1["num"] * r2["den"] + r2["num"] * r1["den"]
    den = r1["den"] * r2["den"]
    # pas de simplification !
    return {"num": num, "den": den}

def egal(r1, r2):
    """Rationnel x Rationnel --> Booleen"""
    # test d'egalite a modifier
    return r1["num"] * r2["den"] == r2["num"] * r1["den"] # produit en croix

def afficher(r):
    """Rationnel --> str"""
    print(str(r["num"]) + "/" + str(r["den"]))
```

On peut vérifier que l'on peut écrire exactement les mêmes instructions que précédemment et obtenir exactement les mêmes résultats, alors même que l'implémentation est totalement différente.

Le programmeur qui utilise une structure de données fait abstraction à la fois :

- de la manière dont les données sont représentées (ex. : l'écriture des instructions et les résultats sont les mêmes, que les données soient représentées par des couples ou des dictionnaires, ou autre chose...)
- de la manière dont les opérations sont programmées (ex. : le test d'égalité ne suit pas la même logique selon les deux implémentations mais le résultat est le même).

III - Structures de données abstraites (SDA)

On parle de structure de données abstraites ou de type abstrait de données car au niveau de l'interface les données, leurs liens et les opérations permises sont caractérisées mais on ne sait pas (et on ne veut pas savoir) comment c'est fait concrètement (implémentation).

Pourquoi avoir recours à cette notion de type abstrait ?

Tout simplement pour définir des types de données non primitifs, c'est-à-dire non disponibles dans les langages de programmation courants. Les types primitifs sont par exemple les entiers, les flottants, les booléens.

Quelques structures de données abstraites

Les structures de données abstraites que nous étudierons cette année peuvent être classées selon la nature de l'organisation de la collection de données :

- structures **linéaires (ou séquentielles)** : il y a un premier élément et un dernier ; chaque élément a un prédécesseur (sauf le premier) et un successeur (sauf le dernier). Exemples : **liste, file, pile**.
- structures **associatives** : les éléments sont repérés par une clé ; ils n'ont pas de lien entre eux. Exemples : **dictionnaires, ensembles**.
- structures **hiérarchiques** : il y a un (parfois plusieurs) élément racine ; chaque élément dépend d'un antécédent (sauf la/les racine/s) et a des descendants (sauf les feuilles). Exemple : **arbre**.
- structures **relationnelles** : chaque élément est en relation directe avec des voisins, ou bien a des prédécesseurs et des successeurs. Exemple : **graphe**.

Ces structures de données sont parfois implémentées nativement dans les langages de programmation comme type de données mais ce n'est pas toujours le cas. En Python, les listes (type `list`), les dictionnaires (type `dict`) et les ensembles (type `set`) le sont mais pas les autres.

Opérations des SDA

Les opérations usuelles d'une SDA sont :

- **Créer une donnée** éventuellement vide, en utilisant ce qu'on appelle un constructeur.
- **Accéder à un élément**
  - → soit ceux directement repérables par la structure (le premier d'une séquence, ou associé à une clé donnée, ...)
  - → soit à partir d'un élément préalablement repéré (successeur d'un élément donné, ...)
- **Ajouter un élément**, en précisant comment il s'intègre dans l'organisation globale de la collection.
- **Retirer un élément**, en précisant comment ceux qui lui étaient liésse réorganisent.
- Eventuellement, des **opérations plus avancées** (rechercher un élément, trier la collection, fusionner deux collections, ...)

Chaque opération doit être bien spécifiée (entrée, sorties, précondition) : c'est ce que l'on fait dans **l'interface**.

Efficacité des SDA

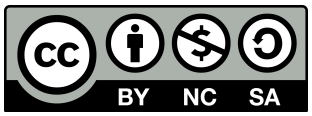
Pour réaliser un traitement algorithmique, les structures de données doivent être choisies :

$\Longrightarrow$ sur le plan de **l'interface** : Comment les données doivent-elles être organisées ? Quels sont leurs liens ? Quelles opérations sont nécessaires pour le traitement ?

$\Longrightarrow$ sur le plan de **l'implémentation** : Quelle est l'implémentation la plus efficace, en temps et en espace, pour le traitement considéré ?

BILAN

- Nous avons vu qu'une structure de données est une méthode de stockage et d'organisation des données destinée à en faciliter l'accès et la modification. Elle regroupe des données à gérer et un ensemble d'opérations qu'on peut leur appliquer.
- Les structures de données s'envisagent à deux niveaux : l'interface (abstrait) et l'implémentation (concret).
  - ★ **L'interface** est la spécification de l'ensemble des opérations de la structure de données. C'est la partie visible pour qui veut utiliser ce type abstrait de données. Elle est suffisante pour utiliser la structure de données.
  - ★ **L'implémentation** consiste à concrétiser - réaliser effectivement - un type de données en définissant la représentation des données avec des types de données existants, et en écrivant les programmes des opérations. L'utilisateur doit pouvoir écrire les mêmes instructions et obtenir les mêmes résultats quelle que soit l'implémentation de la structure de données.
- On peut écrire plusieurs implémentations d'une même structure de données (d'une même interface). Le choix de l'implémentation est en général guidé par les opérations de la structure et leurs coûts pour un traitement algorithmique donné. Nous y reviendrons lorsque nous étudierons quelques structures de données classiques.
- Cette dissociation entre interface et implémentation est présente lorsque nous utilisons une bilbiothèque (ou plus tard dans l'année, une API) : nous n'avons pas besoin de connaître son implémentation pour l'utiliser, son interface suffit. Par exemple, la documentation officielle de la bibliothèque `random`s présente la spécification des opérations disponibles (son interface) et nous suffit pour l'utiliser. Il est néanmoins possible de voir l'implémentation en cliquant en haut de la page sur le lien vers le code source.



Source : Lycée Mounier - Angers