Un peu de didactique de l'informatique

Emmanuel Beffara, d'après Christophe Declercq

Introduction

La pensée informatique

L'expression *computational thinking* introduite par Jeannette Wing fait référence à des compétences et des habiletés humaines, pouvant être développées à l'occasion d'activité de programmation, et transférables à bien d'autres situations de type *résolution de problème*.

Il ne s'agit pas de penser comme une machine, mais de décrire les compétences cognitives en jeu pour faire résoudre un problème par une machine. Il s'agit donc d'activité cognitive de haut niveau et donc bien d'une activité humaine.

NB: *La traduction française fait débat. "Thinking" aurait pu être traduit par "réflexion". Le terme "computationnel" n'existant pas, l'adjectif "calculatoire" est le plus proche mais semble réducteur.*

Les compétences en jeu

L'énumération des compétences en jeu fait débat.

Évaluer attribuer mentalement une valeur à un programme donné.

Anticiper se mettre en posture de programmeur pour décrire l'enchaînement des opérations, avant le début de l'exécution.

Décomposer transformer un problème complexe en un ensemble de problèmes plus simples équivalent au problème initial.

Généraliser inférer un problème général à partir d'une instance, repérer dans un problème particulier la répétition de traitements ou de données suivant un même schéma.

Abstraire "faire abstraction" des informations non pertinentes et créer des solutions où la manière de résoudre un problème peut être "abstraite" à l'aide d'une interface pertinente.

Préambule aux programmes de NSI

Il permet de développer des compétences :

- analyser et modéliser un problème en termes de flux et de traitement d'informations ;
- décomposer un problème en sous-problèmes, reconnaître des situations déjà analysées et réutiliser des solutions ;
- concevoir des solutions algorithmiques ;
- traduire un algorithme dans un langage de programmation, en spécifier les interfaces et les interactions, comprendre et réutiliser des codes sources existants, développer des processus de mise au point et de validation de programmes ;
- mobiliser les concepts et les technologies utiles pour assurer les fonctions d'acquisition, de mémorisation, de traitement et de diffusion des informations ;
- développer des capacités d'abstraction et de généralisation.

- **anticiper** : évoquée en d'autres termes par la formulation *concevoir des solutions algorithmiques*
- **évaluer** : dans les formulations voisines *analyser* et *comprendre*

Évaluer

Définition

Capacité à attribuer mentalement une **valeur** à un programme donné.

Exercice

Que renvoie ce programme?

```
an = 2019
r = an % 4==0 and (not an % 100 == 0 or an % 400 == 0)
```

Exercice

Que renvoie ce morceau de programme?

```
r = an % 4==0 and (not an % 100 == 0 or an % 400 == 0)
```

Posons les choses

Le mot *évaluer* est à comprendre au sens large:

- déterminer le résultat renvoyé
  interprétation = faire le calcul
- déterminer le *type* de donnée renvoyé
  typage, analogue de l'analyse dimensionnelle
- déterminer la fonction qui lie l'entrée à la sortie
  sémantique = description du sens d'un programme

Généralisation

Évaluer la performance d'un programme : compétence de même nature.

Exemple

```
table = [2, 8, 6, 12, 4, 9]
def tri(t):
    for i in range (len(t)):
        for j in range(i, len(t)):
            if t[j] < t[i]:
                t[i], t[j] = t[j], t[i]
tri(table)
```

Évaluer le temps d'exécution, le nombre d'affectations, le nombre de comparaisons. . .

Généralisation

Évaluer la performance d'un programme : compétence de même nature.

Remarques

- Pour une donnée concrète, on peut instrumenter le programme pour lui faire faire cette évaluation en y ajoutant des compteurs, etc. . .
- Pour une donnée générique, on peut aussi faire une évaluation symbolique en fonction d'un paramètre qui caractérise la donnée (sa taille par exemple).

Conclusion

De manière générale, **évaluer** un programme consiste donc à regarder ce programme comme une donnée et à en calculer une valeur par une méthode particulière. Ce changement de plan du programmeur consiste à regarder son programme tel qu'il est, et non tel qu'il aurait voulu qu'il soit.

La compétence *évaluer* est fondamentale pour mettre au point un programme.

Anticiper

Définition

> *Capacité à se mettre dans la posture du programmeur qui doit décrire dans un algorithme l'enchaînement séquentiel/répétitif/conditionnel des instructions, avant même le début de son exécution.*

C'est une compétence fondamentale pour la conception d'algorithmes. C'est aussi un des principaux obstacles didactiques rencontrés par les programmeurs débutants.

« Une propriété difficile à intégrer [. . .] est le caractère différé d'une exécution du programme » J. Rogalski, 1986

Anticiper c'est imaginer

- C'est la part créative du travail du programmeur: imaginer par quel chemin de calculs intermédiaires on peut passer, pour aller des informations disponibles aux informations que l'on souhaite calculer.
- Anticiper les étapes successives du traitement et les différents cas à envisager, c'est tout l'*art de programmer* (cf. Donald Knuth)

Exemple

Pour calculer la médiane d'une série statistique, on pense d'abord à trier cette série, puis on cherche la valeur située au milieu. On doit aussi distinguer les cas où le nombre de valeurs est pair ou impair pour calculer la valeur située au milieu ou la moyenne des 2 valeurs situées au milieu de la série.

Dans l'enseignement

Cette compétence est abordée très tôt dans l'apprentissage de la programmation et l'obstacle associé est dépassé très vite. Cependant:

- toutes les situations problèmes requérant une solution algorithmique font appel à la maîtrise de cette compétence,
- la complexité d'une situation vient de la longueur de la chaine d'anticipation nécessaire et de la nécessité d'inventer des états intermédiaires pour anticiper le passage d'un état initial à un état final.

C'est lié à la compétence **décomposer** car il est souvent utile de profiter d'états intermédiaires identifiés pour séparer le problème.

Importance du modèle d'exécution

Le modèle d'exécution sous jacent et sa compréhension par l'élève ont une influence sur la difficulté à anticiper correctement.

Exemple

L'ordre d'évaluation des expressions et d'exécution des instructions sont des caractéristiques à prendre en compte au moment d'anticiper l'écriture d'un programme.

Dans les programmes

« *Distinguer ce qui est exécuté sur le client ou sur le serveur et dans quel ordre* »
Le modèle d'exécution constitué de deux machines communicantes, est plus difficile que celui d'une seule machine exécutant un programme séquentiel.

Conclusion

La compétence d'anticipation n'est donc pas seulement une capacité d'analyse *descendante* allant du problème vers sa solution algorithmique. Ce peut aussi être une démarche *ascendante* par composition d'instructions en anticipant leur effet pour aller des informations disponibles vers les informations à calculer.

Décomposer

Définition

Capacité à transformer un problème complexe en un ensemble de problèmes plus simples équivalents au problème initial.

- Cette compétence est fondamentale car c'est elle qui permet d'envisager le traitement de problèmes arbitrairement complexes par réduction à des problèmes plus simples ou déjà connus.
- Elle n'est absolument pas spécifique à l'informatique!

Décomposition par cas

À tous les niveaux de la conception d'un programme...

Exemple

Le calcul du nombre de jours d'un mois peut se décomposer en 3 cas distincts: les mois de 31 jours, les mois de 30 jours et février.

Exemple

Le problème de la résolution d'une équation du second degré peut se décomposer en 3 sous-problèmes distincts selon la valeur du discriminant négatif, nul ou strictement positif.

Exemple

Le problème de la compression d'une image peut être séparé en plusieurs sous-problèmes selon le format d'enregistrement de l'image à compresser et selon le souhait de l'utilisateur de compresser avec ou sans perte d'information.

Décomposition séquentielle

Exemple

Programmer la soustraction de deux nombres en binaire, en utilisant le codage en complément à deux, peut se décomposer en 3 étapes:

```
def soustraction_binaire(a, b):
    inv_b = inverser_bits(b)
    moins_b = ajouter_1(inv_b)
    a_moins_b = additionner(a, moins_b)
    return a_moins_b
```

Remarques

On conserve la décomposition suivie. Des variables intermédiaires ont été imaginées et nommées pour contenir les résultats intermédiaires. La solution ne sera complète et exécutable que lorsque toutes les fonctions intermédiaires utilisées auront bien été définies.

Sur la mise en oeuvre

Le résultat de la décomposition d'un problème en problèmes plus simples peut être

- montré explicitement en utilisant une notion de sous-programme / fonction / procédure pour conserver dans le programme écrit la démarche de décomposition suivi,
- caché en recollant ensemble dans un seul traitement toutes les parties élémentaires issues de la décomposition.

La seconde approche est clairement déconseillée car elle donne des fragments de programmes plus complexes et donc plus difficiles à comprendre, puisque le lecteur doit refaire le travail de décomposition.

Conclusion

Savoir décomposer est une compétence permettant d'appréhender des problèmes complexes.

- Sa mise en oeuvre en informatique passe par l'utilisation systématique de primitives de programmation permettant la structuration des programmes:
  - fonctions et procédures,
  - objets et méthodes.
- Ces mécanismes permettent aussi de mettre en oeuvre des démarches de généralisation et d'abstraction.

Généraliser

Définition

> *Capacité à inférer un problème général à partir d'une instance de ce problème, et à repérer dans un problème particulier la répétition de traitements ou de données suivant un même schéma.*

La notion de fonction **avec paramètres** permet de mettre en oeuvre un programme qui peut résoudre un problème général, dont les instances particulières correspondent aux applications de cette fonction avec des valeurs particulières des paramètres.

Exemple, avant généralisation

On cherche à savoir, en heures, minutes et secondes, quelle est la somme des durées 18h20mn30s et 6h45mn50s.

```
total1 = 18 * 3600 + 20 * 60 + 30
total2 = 6 * 3600 + 45 * 60 + 50
total = total1 + total2
mn = total // 60
s = total - mn * 60
h = mn // 60
mn = mn - h * 60
print(h, "h", mn, "mn", s, "s")
```

Première généralisation

Ce sont des calculs analogues qui sont effectués pour `total1` et `total2`. Une première généralisation consiste à définir un traitement générique s'appliquant à trois paramètres et à l'instancier deux fois.

```
def temps (h, mn, s):
    return (h * 3600 + mn * 60 + s)

total1 = temps(18, 20, 30)
total2 = temps(6, 45, 50)
```

Deuxième généralisation

Une deuxième généralisation pourrait alors consister à ne pas traiter le problème uniquement pour l'exemple demandé, mais pour tout temps donné en paramètre.

```
def additionner(h1, mn1, s1, h2, mn2, s2):
    total1 = temps(h1, mn1, s1)
    total2 = temps(h2, mn2, s2)
    total = total1 + total2
    mn = total // 60
    s = total % 60
    h = mn // 60
    mn = mn % 60
    return (h, mn, s)

(h, mn, s) = additionner(18, 20, 30, 6, 45, 50)
print(h, "h", mn, "mn", s, "s")
```

Répétition de schémas algorithmiques : utilisation de fonctions en paramètres

```
def somme(liste):
    resultat = 0
    for elt in liste:
        resultat = resultat + elt
    return resultat
somme([1,2,3,4,5])

def produit(liste):
    resultat = 1
    for elt in liste:
        resultat = resultat * elt
    return resultat
produit([1,2,3,4,5])
```

Fonctions en paramètre

généraliser ces deux fonctions en définissant une fonction `combiner`

```
def combiner(op, neutre, liste):
    resultat = neutre
    for elt in liste:
        resultat = op(resultat, elt)
    return resultat
def addition(x, y):
    return x + y
combiner(addition, 0, [1,2,3,4,5])

def multiplication(x, y):
    return x * y
combiner(multiplication, 1, [1,2,3,4,5])
```

À partir du même schéma, on décline plusieurs instances d'une solution à un problème plus général

Conclusion

- Généraliser est une compétence de haut niveau qui permet au programmeur de résoudre des problèmes plus généraux et ensuite de réutiliser pour des instances particulières des parties de programme déjà écrites.
- La notion de paramètre, utilisée pour instancier des valeurs particulières ou des fonctions particulières est le mécanisme permettant de mettre en oeuvre la généralisation.
- Dans le contexte de la programmation objet, un autre mécanisme permet de prévoir des solutions générales à toute une classe de problèmes: c'est le mécanisme de l'héritage entre classes.

Abstraire

Définition

> *Capacité à "faire abstraction" des informations non pertinentes et à créer des solutions où la manière dont un problème est résolu peut être "abstraite" à l'aide d'une interface pertinente.*

C'est différent de *généraliser*: il n'est pas question d'inférer un problème général à partir d'instances, il est question de formuler des solutions dépendant du minimum d'informations nécessaires.

Abstraire avec les données

Abstraire avec les données consiste à *encapsuler* un certain nombre d'informations en utilisant une structure de données composée qui peut éventuellement masquer le détail du codage des informations.

Exemple

Le programme suivant peut effectuer un calcul de temps si les fonctions intermédiaires utilisées sont définies pour enregistrer les informations sous un format adapté et y accéder. La représentation choisie n'est pas visible à ce niveau. Elle est abstraite. Les informations fournies consistent en des nombres d'heures, minutes et secondes.

```
t1 = temps(18, 20, 30)
t2 = temps(6, 45, 50)
total = additionner_temps(t1, t2)
afficher_temps (total)
```

Exemple, une version : enregistrements

```
def temps(h, mn, s):
    return {'h': h, 'mn': mn, 's': s}
def additionner_temps(t1, t2):
    s = t1['s'] + t2['s']
    mn = t1['mn'] + t2['mn']
    h = t1['h'] + t2['h']
    return {'h':h+(mn+s//60)//60,
            'mn':(mn+s//60)%60, 's':s%60}
def afficher_temps(t):
    print(t['h'], "h", t['mn'], "mn", t['s'], "s")

t1 = temps(18, 20, 30)
t2 = temps(6, 45, 50)
total = additionner_temps(t1, t2)
afficher_temps(total)
```

Exemple, une autre version : tout convertir en secondes

```
def temps(h, mn, s):
    return (h * 60 + mn) * 60 + s
def additionner_temps(t1, t2):
    return t1 + t2
def afficher_temps(t):
    print(t // 3600, "h", (t//60)%60, "mn", t%60, "s")

t1 = temps(18, 20, 30)
t2 = temps(6, 45, 50)
total = additionner_temps(t1, t2)
afficher_temps(total)
```

Pour l'utilisateur, le choix de la structure de données n'est pas nécessairement visible. Le résultat est le même dans les deux mises en oeuvre. D'autres choix de structures de données auraient pu être faits.

Abstraire avec les fonctions

fonctions: masquer la méthode de calcul utilisée

Exemple

Les deux fonctions `fact` ci-dessous calculent bien chacune la factorielle d'un entier positif. Elles sont interchangeables pour l'utilisateur.

```
def fact(n):
    return 1 if n == 0 else n * fact(n - 1)


def fact(n):
    f = 1
    for i in range(n):
        f = f * (i + 1)
    return f
```

Leur écriture sous forme de fonction permet à l'utilisateur de faire abstraction de la manière dont le calcul a été effectué.

Conclusion

- Abstraire en cachant soit les données soit les algorithmes, par des fonctions, permet au programmeur de simplifier l'écriture de ses programmes.
- La combinaison des deux méthodes aboutit à la programmation orientée objet où données et méthodes sont encapsulées à l'intérieur d'une classe.