Université Abdelhamid Mehri – Constantine 2

2023-2024. Semestre 3

Algorithmique et Structures de Données (ASD)

Chapitre 3 : Approche de Résolution Abstraite: Introduction aux Types Abstraits de Données (TAD)

Objectifs du cours

- **Accompagner les étudiants pour apprendre à raisonner de manière abstraite.**
- Initier les étudiants au concept de Type Abstrait de Données (TAD),
  - ✓ **A quoi sert un TAD** ?
  - ✓ **Comment s'en servir**?
  - ✓ **Pourquoi recourir au TAD dans l'élaboration des algorithmes.**
- **Apprendre aux étudiants à construite un TAD en définissant ses données et opérations.**
- **Etre capable d'associer à un TAD, plusieurs implémentations, donc :**
  - ✓ **Plusieurs R.I des données du TAD**
  - ✓ **Plusieurs mises en œuvre des opérations du TAD.**

1. Introduction

Pour concevoir **une solution algorithmique réutilisable**, une **approche abstraite** de résolution de problème est à adopter. L'approche **consiste en une démarche descendante** qui procède par raffinements successifs d'une solution. **Cette approche démarre d'une description abstraite des données et leurs opérations, loin de toute implémentation**. On parle alors de **Type Abstrait de Données (TAD).**

2. Type Abstrait de Données

C'est un moyen de ***définition, de description*** ou ***spécification*** de **données**. Un TAD décrit:

1. la **nature et les propriétés des données,**
2. la **sémantique des opérations,** pouvant être effectuées, sur les données;

indépendement **de de toute implémentation**. C'est-à-dire, loin de toute:

- **Représentation Interne (R.I) de ces données**
- **Mise en oeuvre des opérations (des algorithmes dependants des R.I).**

En général, un TAD admet **différentes implémentations, donc:**

- ✓ **plusieurs représentations internes** possibles des données,
- ✓ plusieurs **algorithmes détaillés (mise en oeuvre)** pour les opérations.

Les avantages des TAD sont:

- Prise en compte de **types de données complexes**.
- **Séparation** entre les **services** (**données + opérations**) et le **codage** (**algorithmes détaillés dependants des R.I**).
- **L'utilisateur d'un TAD** n'a pas besoin de connaître les **détails** du **codage**.
- **Abstraction des services qui favorise la réutilisation**.
- **Ecriture d'algorithmes modulaires, "génériques" réutilisables**.

2.1 Définition d'un TAD

Un **Type Abstrait de Données** (**TAD**) est donc caractérisé par :

- Son **Nom,**
- Les **Sortes** (**domaine de valeurs**) **qu'il manipule,** } ***Signature***
- Les **Opérations sur les Données,**
- Les **Propriétés de ces Opérations.** } ***Axiomes (formules)***

Les trois premières notions définissent **la Signature d'un TAD**, les propriétés sont exprimées généralement sous forme **d'*axiomes (formules)*** dans un TAD.

2.1.1. Signature d'un TAD

Les **sortes** ne sont rien d'autre que des **noms** servant à représenter des **domaines de valeurs** sur lesquels vont porter les opérations. Par exemple: **booleen, entier, date, Etudiant**, **Liste, Pile**, **File** etc.

Chaque **opération** est définie par son **profile** : **les sortes de ses paramètres en entrées et la sorte du résultat**. La **forme** générale du **profile** d'une **opération** n-aire est:

Nom-opération : sorte$_1$, sorte$_2$, ..., sorte$_n$ → sorte$_r$.

Exemple de Signature:

TAD DATE

Sorte : Date

Utilise ENTIER **/TAD réutilisé→ ses sortes et ses opérations sont réutilisées selon le besoin**

Opérations :

définir-date: Entier, Entier, Entier → Date

année: Date → Entier (dit aussi **Naturel**)

mois: Date → Entier

jour: Date → Entier

en-jours: Date → Entier

Fondamental:

- Dans un TAD, on distingue essentiellement les opérations **constructeurs** (une, deux ou plus)**, qui permettent de construire les éléments de la nouvelle sorte,** (celle en cours de définition); **exp:** l'opération **définir-date.**
- Et d'autres opérations qui manipullent les données du TAD et eventuellement ceux des TAD réutilisés.

2.1.2. Axiomes (Propriétés)

La sémantique ou les **propriétés des opérations,** définies dans un TAD, sont exprimées sous formes **d'axiomes (formules).** Les ***axiomes*** **servent donc à donner une signification ou sémantique aux opérations**.

1- **La forme générale de l'écriture des axiomes** (**formules**) est:

$$\mathbf{terme}_{\mathbf{gauche}} \equiv \mathbf{terme}_{\mathbf{droit}}$$

Le symbole "≡" se lit **est "équivalent en sens à"**

2- **Un axiome peut aussi s'écrire sous une forme conditionnelle** comme suit:

$$\mathbf{terme}_{\mathbf{gauche}} \equiv \textbf{ si condition alors } \mathbf{terme}_{\mathbf{droit1}} \textbf{ sinon } \mathbf{terme}_{\mathbf{droit2}}$$

Fondamental:

1. $\mathbf{Terme}_{\mathbf{gauche}}$ et $\mathbf{Terme}_{\mathbf{droit}}$ **sont des opérations du TAD**
2. Nous définissons des **d'axiomes (formules)** pour les opérations **non constructeurs.**
3. **Le nombre** d'**axiomes pour chaque** opération **non constructeur est égale au nombre d'opérations constructeurs**.

Remarque:

1. **La forme conditionnelle donne des fois des axiomes récursifs.**
2. Pour **simplifier la définition de la sémantique des opérations, vous pouvez donner une formule itérative («axiome itératif »)** pour vos opérations.

Exemples d'axiomes (formules)

Variables: /* variables au sens mathématique.
x, y, z : Entier;
d, d' : date.

Axiomes :

Année (définir-date(x,y,z)) ≡ z

Mois (définir-date(x,y,z)) ≡ y

Jour (définir-date(x,y,z)) ≡ x

en-jours (définir-date(x,y,z)) ≡ x + y*30 + z * 365

Important : L'opération en-jours peut s'écrire d'une manière qui favorise la réutilisation des opérations, c.a.d des axiomes à utiliser directement dans l'implémentation (niveau fonctions).

en-jour(d) ≡ jour(d) + mois(d)*30 + année(d)*365

Remarque

Dans **en-jour(d) : la variable d, de sorte Date, a remplacé le constructeur définir-date(x,y,z) dans la partie gauche, d'où, la partie droite de cet axiome (qui est : jour(d) + mois(d)*30 + année(d)*365) formera le corps de la fonction en-jour(d) (à l'implémentation du TAD).**

2.1.3. Précondition des opérations

Une Précondition d'une opération Op1 est une condition à tester avant l'appel de la fonction fonc1 associée à Op1.
Exemple: Précondition pour l'opération depiler (P) est: depiler (P) est définie ssi est-vide(P)=faux

3. Construction d'un TAD

Nous allons montrer l'intérêt de **décrire les Structures de Données (S.D) à utiliser dans un algorithme avec un TAD** à travers l'activité ci dessous. Nous montrons aussi comment passer d'une telle **description à son implémentation** (pseudo-code **en algorithmique**) ou sa **codification** dans **n'importe quel langage de programmation**.

Activité 1
Considérons les données: "nombres complexes". construisez un **TAD qui décrit la structure de données " nombres complexe"**

TAD COMPLEXE
Utilise REEL
Sorte complexe
Opérations
constC : réel, réel → complexe Opération **constructeur**

rl : complexe → réel
im : complexe → réel
module: complexe → réel

addC : complexe, complexe → complexe
soustC : complexe, complexe → complexe
.... **// Nous pouvons tjs enrichir un TAD par d'autres opérations.**

Variables
r1, r2, r3, r4 : réel,
z, z1, z2 : complexe
Axiomes

1- rl(constC(r1,r2)) ≡ r1 **// Constructeur en partie gauche**

2- im(constC(r1,r2)) ≡ r2

3- module((constC(r1,r2)) ≡ sqrt(r1*r1 + r2*r2)) //+, * et sqrt sont définies dans le TAD REEL

4- addC(constC(r1,r2), constC(r3, r4)) ≡ constC(r1+r3, r2+r4)

5- soustC(constC(r1,r2), constC(r3, r4)) ≡ constC(r1-r3, r2-r4)

On peut écrire d'une autre manière ces 3 derniers axiomes: (cette manière favorise la réutilisation de l'axiome à l'implémentation. Elle consiste à utiliser des **variables dans la partie gauche de l'axiome** et **non des opérations constructeurs.**

module(z) ≡ sqrt(rl(z)*rl(z) + im(z)*im(z)))

addC(z1, z2) ≡ constC(rl(z1)+rl(z2), im(z1)+im(z2))

soustC(z1, z2) ≡ constC(rl(z1)-rl(z2), im(z1)-im(z2))

Important: **Cette 2eme manière n'utilise pas les constructeurs ds les arguments des opérations** (**partie gauche de l'axiome, mais des variables de la sorte définie**, **ici Z, Z1 et Z2: complexe**) **et elle permet ainsi d'utiliser l'axiome à l'implémentation (voir le 4. ci dessous). C'est la forme des axiomes qu'il faut tjs assayer de trouver.**

4. Implémentation d'un TAD

Maintenant, pour **implémenter un** TAD, donc lui associer**:**

1. une **Representation Interne (R.I)** ou **un Type** et**,**
2. des **fonctions** pour **les operations,**

il suffit d'appliquer les correspondances résumées dans le tableau suivant :

Concepts dans un TAD (ou Eléments du TAD)	**Implémentation en pseudo-code ou en langage de programmation**
Sorte	Soit un: • **Type de base du langage (int, real, char, tableau, enreg)** • **Type défini par le concepteur.**

Opération non constructeur • **Profile** • **Axiome**	**Fonction** • **Entête** • **Corps (instructions tirées de l'axiome)** **des fois le corps est identique à la partie droite de l'axiome**
Opération constructeur • **Profile**	**Fonction** • **Entête** • **Corps (instructions) est écrit selon le type implémentant la sorte définie dans le TAD**
Précondition d'une opération	**Condition d'appel de la fonction, à tester avant l'appel de la fonction**

Ainsi, si nous voulons **implémenter le TAD COMPLEXE**, il suffit d'associer:

1. une **Représetation Interne (R.I)** à la sorte **complexe,** qui peut être:
   a) **un enregistrement à deux champs pour la partie réel et imaginare.**
   b) un **tableau à deux éléments**)
   c) ou deux **variables séparées**.
2. et une **implémentation adéquate, donc des fonctions** pour les différentes **opérations définies** dans le TAD.

L'implémentation proposée, ci dessous, repose sur **la R.I des nombres complexes par un enregistrement.**

TAD COMPLEXE	**Implémentation en pseudo code**
Sortes • réel • complexe	**Types** • **REAL** • **complexe = Enregistrement** **R, M: real** **Fin**
Opérations non constructeurs **profile** **addC: complexe, complexe → complexe** **Axiome** **addC(z1,z2) ≡ constC (rl(z1)+rl(z2), im (z1)+im(z2))** **profile** **soustC : complexe, complexe → complexe**	**Fonction addC (z1, z2: complexe): complexe** Déclarations / declarez toutes les fonc utilisées **Fonction rl (z:complexe): real** debut ... fin **Fonction im (z:complexe): real** debut ... fin **Fonction constC (r1, r2:real): complexe** debut ... fin **Début** **Retourner (constC (rl(z1)+rl(z2), im(z1)+im(z2))** **Fin** **Fonction soustC (z1, z2: complexe): complexe** Déclarations **Fonction rl (z:complexe): real** debut ... fin **Fonction im (z:complexe): real** debut ... fin **Fonction constC (D r1, r2:real): complexe** debut ... fin

Axiome **soustC(z1,z2) ≡ constC(rl(z1)-rl(z2), im(z1)-im(z2))**	**Début** **Retourner (constC (rl(z1)-rl(z2), im(z1)-im(z2))** **Fin**
profile **module: complexe→ réel**	**Fonction module (z: complexe): real** Déclarations **Fonction rl (z:complexe): real** debut ... fin **Fonction im (z:complexe): real** debut ... fin **Début** **Retourner (sqrt (rl(z)*rl(z) + im (z)*im (z)))** **Fin**
Axiome **module(z) ≡ sqrt (rl (z)*rl(z) + im (z)*im(z)))** **//axiome utilisable directement en implémentation**	
profile **rl : complexe → réel** **Axiome** **rl(constC(r1,r2)) ≡ r1 /*cet axiome ne peut etre utilisé directement ds l'implémentation (mais il a définit le sens de l'operation)**	**Fonction rl (z:complexe): real** **Début** **Retourner (z.R) /*pas d'axiome à utiliser directement, d'ou l'implementation dépend du type enreg** **Fin**
profile **im : complexe → réel** **Axiome** **Im (constC (r1,r2)) ≡ r2 //mm chose pr l'op: Im**	**Fonction im (z:complexe): real** Début **Retourner(z.M)** Fin
constC : réel, réel → complexe pas d'axiome pour les constructeurs, alors le corps de la fonction s'écrit à l'aide d'instructions et le type déclaré.	**Fonction constC (r1, r2: real): complexe** Déclaration **z: complexe** **// l'op constructeur est implémentée avec le type déclaré, ici enregistrement** **Début** **z.R=r1** **z.M=r2** **Retourner(z)** **Fin**

Remarque

1-Notons que parmi les fonctions implémentant les opérations du TAD COMPLEXE, certaines sont dites de base (telles que: constC, rl et Im), leur corps dépend du type choisi pour implémenter la sorte en question, et d'autres sont valables quelque soit le type considéré (telles que: addC, soustC, module, etc). En changeant de type pour complexe (passer de l'enregistrement au tableau par exemple), ces fonctions restent inchangées et independantes des types declarés (aspect réutilisation de ces fonctions).

2- Ici, nous avons développé des modules (des fonctions) séparés, à exploiter ds un algorithme complet.

3- Pour une bonne assimilation des concepts: TAD et leur implémentations, voir les vidéos 9, 10 et 12.

Test de sortie:

En se basant sur le TAD COMPLEXE, étudié en cours, vous avez un devoir maison en algorithmique à compléter par une programmation en TP (durée du TP: 1 semaine), pour:

1. ***Developpez un algorithme Complexe-abstrait, composé de procédures et fonctions, qui permet de:***

   1.1. Créer 2 nombres complexes.
   1.2. Afficher leur somme et soustraction.
   1.3. Donner le nombre complexe, dont le module est le plus grand.

2. ***Si vous optez pour une R.I (*****Representation Interne*****) des nombres complexes à l'aide d'un tableau Comp de 2 réel, quelles fonctions de l'algorithme précédent faut il adapter à la R.I choisie?***

3. ***Donnez le code translatant, l'algorithme Complexe-tableau, en un programme Java (bien sur aprés avoir instancier l'algorithme Complexe-abstrait par l'algorithme Complexe-tableau).***

NB:

- ✓ ***l'objectif de ce travail est de produire un algorithme "abstrait", puis l'instancier par un algorithme "concret", mais comportant le plus possible des fonctions "génériques" du premier algorithme. Puis translater le second algorithme en un programme Java.***

- ✓ ***L'évaluation du TP, concerne essentielllement la vérification de la réutilisation des fonctions "génériques".***

Bonne compréhention de l'approche abstraite et ses pratiques en TD et TP.