INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES RENNES

# Structures de données

## Chapitre 1 : Les types abstraits de données (TAD)

Yann.Ricquebourg@insa-rennes.fr - Version 22.0920





## Présentation du module

- **« Structures de données » ?**
  - **Ce sont les éléments logiciels pour stocker des données**
  - **Il y en a de nombreuses déjà conçues, certaines sont la base**
    - utiles à connaître (pour l'utilisation)
    - utiles à décortiquer (pour savoir ce qu'il y a « sous le capot »)

- **Organisation générale**
  - **8 chapitres dédiés à différents cas d'usage**
  - **Sujets de TP associés au fil des chapitres**
  - **Fonctionnement avec partie « classe renversée »**
    - 6 équipes d'étudiants => pour préparer la présentation de 6 chapitres
    - Inscription en équipe sur Moodle
    - 1 séance de préparation interactive avec l'équipe concernée quelques jours avant l'exposé durant le cours sur ce chapitre
    - Cf. notes rédigées en fiches fournissant les idées directrices (sur Moodle)

      => idéalement en prendre connaissance (ainsi que du sujet de TP) avant la séance de préparation interactive avec l'enseignant

Types abstraits de données

# 1 ► Introduction



## Introduction

- **Pourquoi abstraire ?**
  - pour mieux comprendre
  - pour avoir des solutions génériques et réutilisables
  - pour pouvoir démontrer ou vérifier des propriétés
- **Comment abstraire ?**
  - Bien identifier les préoccupations
  - Séparer les rôles
    - notamment le rôle de spécification et le rôle d'implémentation
    - c.-à-d. bien séparer la définition d'utilisation de la future mise en œuvre
  - Utiliser des modèles pour respecter un cadre
- **→ Les TAD vont fournir une réponse, un cadre formel**

Types abstraits de données

# 2 ► Séparer la mise en œuvre de l'utilisation



## Séparer utilisation et mise en œuvre

- **Est-ce le rôle assuré par une interface ?**
  - permet plusieurs mises en œuvre
  - un utilisateur peut les utiliser sans connaître les détails de leurs implémentations
  - = concept de « boîte noire »

- **Exemple Java :**

```
/** Une pile d'entiers. */
public interface Pile {
        void empiler(int v);
        void depiler();
        int sommetPile();
        boolean pileVide();
}
```

  - on a juste spécifié des noms, mais concernant l'utilisation il manque :
    - la spécification des actions, les modifications engendrées sur l'état
    - la spécification du résultat rendu par les interrogations sur l'état

# Séparer utilisation et mise en œuvre



- **Ce qui n'est pas spécifié (mais c'est notre but) :**
  - comment cet état est conservé
  - = comment réaliser l'implémentation

- **Exemple de choix de mise œuvre Java (implémentation) :**

```
public class PileTabulée implements Pile {
        private static final int N = 100;
        private int[] tab;
        private int pointeur; // indique la prochaine case à remplir

        public PileTabulée() {
                tab = new int[N];
                pointeur = 0;
        }

        public boolean pileVide() {
                return (pointeur == 0);
        }
```

INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES RENNES

# Séparer utilisation et mise en œuvre



- **(suite)**

```
        public void empiler(int v) {
                if (pointeur != N) {
                        tab[pointeur] = v;
                        pointeur++;
                }
        }

        public void depiler() {
                if (!pileVide())
                        pointeur--;
        }

        ...
}
```

- **Exercice : comment s'écrit la méthode qui manque ?**

# Séparer utilisation et mise en œuvre



- **Étant donné cette interface**
  - très bien : on n'explicite aucune implémentation concrète
    - on peut choisir d'implémenter autrement
      par exemple tout simplement en changeant le sens du remplissage
    - l'interface est donc bien séparée de la mise en œuvre
  - moins bien : rien ne garantit ce que l'implémentation doit respecter
    - on peut implémenter n'importe quoi
    - tout au mieux il y aura des commentaires dans l'interface pour documenter le résultat souhaité, mais avec leur lot d'ambiguïtés (langage naturel) et que le programmeur peut mal comprendre



# Séparer utilisation et mise en œuvre



- **Une autre mise en œuvre conforme à l'interface**

```
public class PileOuFace implements Pile {
        private boolean face;

        public PileOuFace() {
                face = false;
        }
        public boolean pileVide() {
                return face;
        }
        public void empiler(int v) {
                face = (v>0);
        }
        public void depiler() {
                face = !face;
        }
        public int sommetPile() {
                if (face) return 1;
                else return 0;
        }
}
```

# Une interface est insuffisante



## Conclusion

- Évidemment le futur utilisateur de PileOuFace sera surpris
  - Le comportement n'est pas celui attendu pour une Pile
  - Pourtant cette mise en œuvre respecte l'interface Pile
- → une simple interface ne garantit rien sur la sémantique de l'opération réalisée
- → il faut un formalisme plus étendu





Types abstraits de données

# 3 ► Vers des spécifications formelles

# Des spécifications formelles

- **Définition (informelle !)**
  - On voudrait une notation complétement définie
    - syntaxiquement et sémantiquement
  - Pour ne laisser aucune ambiguïté dans la spécification
    - c'est la différence entre du langage courant et un formalisme
  - Par exemple
    - « beaucoup de fois un petit peu » => ça se discute
    - 1 000 000 x 1 donne 1 000 000 => clair et sans ambiguïté en maths.
  - Ensuite, on pourra bâtir une mise en œuvre qui respecte le but escompté formalisé par cette spécification
    - comme un plan d'architecte
    - indépendant de la mise en œuvre ➔ ne décrire que le résultat



# Spécification au-delà des interfaces Java

- **Interfaces Java insuffisantes**
  - ➔ que proposer à la place ?
- **1re idée : spécification par implémentation**
  - problème : propre à un cas particulier

# Spécification au-delà des interfaces Java

- 2e idée : spécification par l'exemple
  - problème : combien de petits schémas pour être complet ?



# Spécification au-delà des interfaces Java

- 3e idée : spécification par états
  - problème : toujours pas très formel !

# Spécification au-delà des interfaces Java

- **4e idée : spécification algébrique**
  - ce qui compte est « ce que ça fait » et non « comment c'est fait »
  - → on veut donc spécifier les comportements vis-à-vis de l'état de l'objet
  - → on va utiliser pour cela le formalisme général des fonctions (maths)
    - pour exprimer le résultat
    - en fonction de variables d'entrée
    - et de l'objet lui-même
- **Il y aura**
  - des fonctions
    - dépiler(p), empiler(p, x), etc.
    - **= guides pour les signatures des opérations à implémenter (interface)**



# Spécification au-delà des interfaces Java

- **Il y aura aussi**
  - des préconditions
    - **= guides pour détecter les cas d'erreurs d'utilisation**
    - mais ne donnent bien sûr pas les erreurs liées au choix d'implémentation

      par exemple pour la pile elles spécifieront :

      - l'impossibilité de dépiler dans une pile vide,

      - mais pas le problème d'empiler dans une pile qui serait stockée dans un tableau limité

      → c'est l'implémentation qui rajoute une limite absente du concept abstrait
  - des axiomes
    - ce sont des vérités de base, élémentaires, non démontrables

      par exemple : sommetPile(empiler(p, x)) = x
    - **= guides pour l'implémentation et pour rédiger des tests**

Types abstraits de données

# 4 ▶ Le formalisme des TAD



## Le formalisme des TAD

- **Les Types Abstraits de Données (ou Types Abstraits Algébriques)**
  - (ADT, *Abstract Data type,* en anglais)
  - **sont définis en termes d'opérations abstraites**
    - sans référence à une mise en œuvre particulière
    - sans référence explicite à un état interne courant
  - **spécifient successivement :**
    - les autres types utilisés
    - la signature de leurs fonctions
    - les préconditions pouvant exister pour ces fonctions
      = restriction du domaine de définition
    - les axiomes nécessaires et suffisants pour définir toute la sémantique du TAD

# Le formalisme des TAD : un exemple

## Exemple complet : TAD Pile

**SORTE** Pile
**UTILISE** entier, booléen
**FONCTIONS**
new : ⟶ Pile
pileVide : Pile ⟶ booléen
empiler : Pile × entier ⟶ Pile
dépiler : Pile –/→ Pile
sommetPile : Pile –/→ entier
**AXIOMES**
∀ Pile p, ∀ entier x
sommetPile(empiler(p, x)) = x
dépiler(empiler(p, x)) = p
pileVide(new) = vrai
pileVide(empiler(p, x)) = faux
**PRÉCONDITIONS**
pré(dépiler(p)) = non pileVide(p)
pré(sommetPile(p)) = non pileVide(p)



# Le formalisme des TAD : classifications

## Remarque

- Fonctions partielles
  - Les fonctions ayant une précondition (non vide) sont dites partielles
  - Elles sont dénotées en barrant la flèche du domaine de définition

## Classification des fonctions

- créateur : TAD uniquement à droite de la flèche (et rien à gauche)
- commande : TAD présent des 2 côtés

  → au moment de la mise en œuvre, le programmeur pourra choisir de l'implémenter de 2 façons :
  - producteur : crée un résultat en fonction de l'entré
  - mutateur : transforme l'état interne de l'entrée
- requête : TAD uniquement à gauche
  - = observateur

# Le formalisme des TAD : démonstrations



- **Que faire d'autre avec les TAD ? Démontrer !**
  - **Théorème :**
    - propriété que l'on peut établir à l'aide des axiomes ou d'autres théorèmes déjà établis
- **Fonctionnement déductif des théorèmes**

**→ on peut montrer que $t_1 = t_2$ par transformation d'un des 2 termes**

- **par exemple : on peut remplacer un terme $t$ par un autre $t'$ si $t = t'$ existe en axiome, ou théorème déjà démontré**
- **autres règles de transformation usuelles :**
  - **Symétrie** : si $t_1 = t_2$ alors $t_2 = t_1$
  - **Transitivité** : si $t_1 = t_2$ et $t_2 = t_3$ alors $t_1 = t_3$
  - **Substitution** : soient $t_1[x] = t_2[x]$ de même sorte à variable x dans X si u expression de même sorte que x, alors $t_1[u/x] = t_2[u/x]$
    « on peut remplacer x par l'expression u »
  - **Congruence** : si $t_1 = t_1'$, $t_2 = t_2'$, … , $t_n = t_n'$ alors $f(t_1, \dots, t_n) = f(t_1', \dots, t_n')$
    « on peut appliquer une fonction de part et d'autre »

INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES RENNES

# Le formalisme des TAD : démonstrations



- **Que faire d'autre avec les TAD ? Démontrer !**
  - **Exemple :** démontrer que pileVide(dépiler(empiler(new, e))) = VRAI

    - Partons de
      dépiler(empiler(p, e)) = p
    - Par substitution de p par new
      dépiler(empiler(new, e)) = new
    - Par congruence
      pileVide(dépiler(empiler(new, e))) = pileVide(new)
    - Or pileVide(new) = VRAI, donc par transitivité
      pileVide(dépiler(empiler(new, e))) = VRAI  CQFD.

# Le formalisme des TAD : évaluations

- **Que faire d'autre avec les TAD ? Évaluer !**
  - **par réécriture grâce aux axiomes**
    - sommetPile(empiler(empiler(dépiler(empiler(new, 0)), 1), 2)) = ?
    - on pose p = partie soulignée
    - d'après le 1er axiome de Pile on déduit immédiatement = 2
- **Remarque**
  - **il faut toutefois vérifier que l'expression p « est bien une pile »**
  - **c.-à-d. respecte les règles spécifiant la sorte Pile**



# Le formalisme des TAD : évaluations

- **Que faire avec les TAD ? Évaluer !**
  - **par « exécution » des opérations du TAD**
    - sommetPile(empiler(empiler(dépiler(empiler(new, 0)), 1), 2)) = ?

# Le formalisme des TAD : correction

- **Comment savoir si un TAD est correct ?**
  - **démontrer qu'un TAD est parfaitement correct par rapport à l'idée de son concepteur n'est pas possible dans tous les cas**
    - pas de recette miracle systématique hélas !
  - **inversement, on y trouver des problèmes :**
- **Y a-t-il des contradictions ?**
  - **Trouver un exemple qui contredit un axiome ou théorème**
    - Exemple : si on introduit un axiome de symétrie dans notre TAD Pile
      dépiler(empiler(p, e)) = empiler(dépiler(p), e))
      on prouverait facilement que ce TAD est faux (avec un contre-exemple)
- **Y a t-il convergence ?**
  - **Deux réécritures dans un ordre différent doivent donner le même résultat**
- **Y a-t-il complétude du TAD ?**
  - **Peut-on réécrire toute expression pertinente, si non il manque des axiomes**
    - Exemple : on peut réussir à montrer (par récurrence) que la taille de l'expression réécrite diminue



Types abstraits de données

# 5 ► Du TAD à la mise en œuvre

# Du TAD à la mise en œuvre (Java)



## Signatures des fonctions du TAD

- Permettent de traduire en interface Java
- Attention :
  - traduction aussi d'un monde de fonctions à un monde d'objets & méthodes
  - la fonction empiler(p,x) devient l'appel de méthode p.empiler(x)

## Remarques

- Pas de constructeurs dans les interfaces Java :
  - pas de traduction des créateurs dans l'interface
- Réfléchir à la traduction des commandes :
  - distinguo pratique entre producteurs (return) et mutateurs (void, état interne)
  - la logique et l'aspect pratique guident le choix du programmeur

    Exemple : dépiler(p)

    plutôt : `public void depiler() {...}`

    ou bien : `public Pile depiler() {...}`



# Du TAD à la mise en œuvre (Java)



## Préconditions des fonctions du TAD

- Permettent de détecter les erreurs d'utilisation

## Programmation par contrat

- Ne pas intégrer ces tests d'erreur dans la programmation
  - L'appelant se doit d'appeler nos fonctions dans leur domaine de définition
  - On doit donc documenter les restrictions liées aux préconditions

    L'utilisateur devra « respecter ce contrat »

  - → code plus léger, mais risqué
  - → préférer la programmation défensive

# Du TAD à la mise en œuvre (Java)



- **Programmation défensive**
  - La vérification de validité est effectuée en interne aux opérations

- **via l'introduction de « valeurs spéciales »**
  - Renvoyer une valeur spécifique quand une erreur est détectée
    - Exemple : NaN (*Not A Number*) des fonctions mathématiques en langage C
  - Ces nouvelles valeurs devraient alors être spécifiées par le TAD et prises en compte dans les axiomes (lourd !)

- **via le mécanisme des exceptions (disponibles en Java)**
  - Traduire toute précondition par une vérification pouvant lever une exception
    - Exemple :

```
public int sommetPile() throws PileVideException {
  if (this.pileVide()) {
    throw new PileVideException("Impossible de dépiler une pile vide");
  }
  // Poursuite du traitement non exceptionnel
  ...
```

INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES RENNES

# Du TAD à la mise en œuvre (Java)



- **Axiomes des fonctions du TAD**
  - permettent d'écrire des tests
    - tous les axiomes doivent être vérifiés dans les tests
  - guident la sémantique de l'implémentation des méthodes
    - permet au programmeur de « comprendre » sans ambiguïté le résultat que doit produire son implémentation

Types abstraits de données

# 6 ► Mot de la fin sur les TAD



## Généricité des TAD

### TAD génériques ?

- **Motivation :**
  - il existe des piles d'entiers, piles de réels, piles d'assiettes, etc.
  - → ne pas être obligé de définir autant de types abstraits que de cas d'utilisation alors qu'ils obéissent au même principe au même modèle

- **Solution : la généricité**
  - paramétrer le TAD par une sorte auxiliaire

    par exemple Pile[T]
  - regrouper et décrire en un seul modèle toute une famille de cas particuliers similaires par l'introduction d'un paramètre formel, ici T

# Exemple

## Exemple complet : TAD Pile générique

**SORTE** Pile[T]
**UTILISE** booléen
**FONCTIONS**
new : ⟶ Pile[T]
pileVide : Pile[T] ⟶ booléen
empiler : Pile[T] × T ⟶ Pile[T]
dépiler : Pile[T] –/→ Pile[T]
sommetPile : Pile[T] –/→ entier
**AXIOMES**
∀ Pile p, ∀ T x
sommetPile(empiler(p, x)) = x
dépiler(empiler(p, x)) = p
pileVide(new) = vrai
pileVide(empiler(p, x)) = faux
**PRÉCONDITIONS**
pré(dépiler(p)) = non pileVide(p)
pré(sommetPile(p)) = non pileVide(p)



# Interface correspondante

## Interface Java correspondant à l'exemple

```
public interface Pile<T> {

        void empiler(T v);
        void depiler() throws PileVideException;
        T sommetPile() throws PileVideException;
        boolean pileVide();

}
```

# Conclusion



- **TAD utilisés ?**
  - **Approche formelle non ambiguë séduisante**
  - **Séparation spécifications / mise en œuvre parfaitement atteinte**
  - **Pas utilisables tous les jours**
    - notamment particulièrement lourd d'exprimer tous les axiomes liant les fonctions d'un TAD quand elles sont nombreuses.
    - « *les TAD sont parfaits pour spécifier… les piles !* » ☺

INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES RENNES



**THE END**

**(FIN)**