SDD

1

Structures de Données.

I] Types abstraits.

① Signature : décrit la syntaxe du type abstrait :

- nom des opérations.
- type des arguments.

Type vecteur : opérations : ↳ i-ème : Vecteur x Entier → Element.

↳ i-ème : Vecteur x Entier → Element
↳ changer i-ème : Vecteur x Entier x Element → Vecteur.
↳ bornesup : Vecteur → Entier
↳ borne inf : Vecteur → Entier.

Propriétés du type abstrait : Axiomes.

② Hiérarchie des Types.

↳ Opération interne : opération dont le résultat est du type qui est défini.
(changer i-ème)

↳ Observateur : opération dont le résultat est de type prédéfini ou d'un type abstrait précédemment défini. (ex: i-ème, bornesup, borneinf).

③ Des types aux Classes (Java)

- On associe à chaque Type une Classe.
- Opération du Type ↔ méthode de la Classe.
- Traduction des opérations : - premier argument de l'opérateur ↔ receveur
  - autres arguments ↔ paramètres de la méthode.

④ Propriétés (Description)

Axiomes :

$borneinf(v) \leq i \leq bornesup(v)$
$\Rightarrow$ ième (changerième $(v, i, e), i) = e$

$(bornеinf(v) \leq i \leq bornesup(v) \wedge borninf(v) \leq j \leq bornesup(v) \wedge$
$i \neq j) \Rightarrow$ ieme (changerième $(v, i, e), j) =$ ième (v, j)

Spécification algébrique = Signature + axiomes.

→ Une spécification algébrique doit être $\boxed{\text{consistante}}$; c'est-à-dire, l'ensemble des axiomes n'est pas contradictoire.
→ A-t-on écrit suffisamment d'axiomes ? problème $\boxed{\text{complétude}}$
$\Rightarrow$ complétude suffisante.
il faut écrire des axiomes définissant les observateurs sur les fonctions internes en tenant compte du domaine de définition des observateurs.

On ajoute donc : Vect : Entier × Entier ⟶ Vecteur.
init : Vecteur × Entier ⟶ Booléen.

Précondition :

ième(v, i) est défini si $borneinf(v) \leq i \leq bornesup(v) \wedge init(v, i) =$ vrai.

SDD

2

Axiomes :

init (vect(i, j), k) = faux

borneinf(v) ≤ i ≤ bornesup(v) ⇒ init(changer-ième (v, i, e), i) = vrai

borneinf(v) ≤ j ≤ bornesup(v) ∧ i ≠ j (∧ borneinf(v) ≤ i ≤ bornesup(v))
⇒ init (changer-ième (v, i, e), j) = init (v, j)

borneinf(vect(i, j)) = i

borneinf(changer-ième (v, i, e)) = borneinf(v)

bornesup(vect(i, j)) = j

bornesup(changer-ième (v, i, e)) = bornesup(v)

• **Exercice :** Définir le type abstrait paramétré Ensemble [T].

Type : Ensemble [T].

Opérations :

constructeurs de base
- vide : . ⟶ Ensemble [T]
- ajouter : Ensemble [T] × T ⟶ Ensemble [T]

observateurs
- cardinal : Ensemble [T] ⟶ Entier
- appartient : Ensemble [T] × T ⟶ Booléen

- union : Ensemble [T] × Ensemble [T] ⟶ Ensemble [T]
- intersection : Ensemble [T] × Ensemble [T] ⟶ Ensemble [T]

Précondition :

ajouter (ens, el) défini ssi appartient (ens, el) = faux

Axiomes :
- cardinal(vide()) = 0
- cardinal (ajouter (ens, el)) = cardinal(ens) + 1.
- appartient (vide(), el) = faux.
- appartient (ajouter (ens, el), el) = vrai.
- el1 ≠ el2 ⇒ appartient (ajouter(ens, el1), el2) = appartient (ens, el2).

- union(ens, vide()) = ens

- cardinal (union(ens1, ens2)) =
  cardinal(ens1) + cardinal(ens2) − cardinal(intersection(ens1, ens2)).
- appartient(union(ens1, ens2), el) =
  appartient(ens1, el) ∨ appartient(ens2, el).
- appartient(intersection(ens1, ens2), el) =
  appartient(ens1, el) ∧ appartient(ens2, el).
- cardinal (intersection(ens1, ens2)) = ---
- cardinal (intersection (ens, vide()) = 0
- cardinal (intersection (ens1, ajouter(ens2, el))) =
  | cardinal (intersection (ens1, ens2)) si appartient (ens1, el) = faux
  | cardinal(intersection (ens1, ens2)) + 1 si appartient (ens1, el) = vrai

⑤ Tests des implémentations de types abstraits.

- tests : " techniques boîtes noires ". On s'intéresse au comportement des fonctions. (indépendantes de l'implantation).
- tests : " techniques boîtes blanches " : propre à chaque implantation (non traité dans ce cours)

Problématique :

- optimisation par rapport aux implantations
- couverture des tests
- optimisation au niveau de la lecture des tests

⇒ produire des messages d'erreur

SDD
3

implantation :

interface Ensemble

ArrayEnsemble   VectorEnsemble.

Test Ensemble
(abstract class).

setUpVide()
setUpUnElement()
setUpQuelconque()
TestArrayEnsemble

setUpVide()
setUpUnElement()
setUpQuelconque()
TestVectorEnsemble

Dans TestEnsemble, on met en commun la structure générale des tests.
Pour le type Ensemble dans TestEnsemble :

```
abstract class TestEnsemble {

    public void testAppartient() {
        // tester les axiomes concernant la fct appartient()
    }
    public void testCardinal() {
        //idem pour la fct cardinal()
    }
    public static void main (String[] args) {
        // à compléter
    }
```

Test d'un axiome : "gauche = droite."

- construction de "gauche" : (on construit des objets ...)
- construction de "droite" : (on construit des objets ...)

ex: appartient (vide(), el) = faux.

on construit l'ensemble vide grâce à une méthode :

setUpVide() → abstraite dans TestEnsemble
setUpVide() → implantée dans TestArrayEnsemble et TestVectorEnsemble

→ appartient (ajouter (ens, el) , el) = vrai.

Pour les ensembles on distingue 3 classes d'équivalence :

- ensemble vide
- ensemble à 1 seul élément
- ensemble comportant strictement plus d'un élément.

Principe : 1 Un objet utilisé pour tester un axiome ne doit pas être réutilisé pour tester un autre axiome.

Problème : gauche (ens) = droite (ens)

cardinal (supprimer (ens, el)) = cardinal (ens) - 1 .

Solution : - Dans ce cas, il faut dupliquer l'objet ens
- ou "inverser" l'axiome.

Exercice : Spécification :

Une application commerciale doit gérer un tarif , c.à.d. une collection de produits étiquetés par des prix , en utilisant différentes opérations permettant de

- s'assurer de l'existence d'un produit dans le tarif.
- connaître le prix d'un produit existant dans le tarif.
- connaître le nb produits du tarif
- supprimer un produit du tarif.
- ajouter un nouveau produit avec son prix

① Ecrire la spécif. algébrique du type Tarif (signature + préconditions + axiomes).

② Ecrire un programme de test d'une implantation de Tarif en s'appuyant sur la spécification algébrique.

SDD
4

① Type Tarif

Opérations :

Opérations internes — observateurs

nouveau : $\longrightarrow$ Tarif -- créer un Tarif vide
ajouter : Tarif × Produit × reel $\longrightarrow$ Tarif.
existe : Tarif × Produit $\longrightarrow$ Booléen
nbProduits : Tarif $\longrightarrow$ Entier
supprimer : Tarif × Produit $\longrightarrow$ Tarif.
prix : Tarif × Produit $\longrightarrow$ Reel.

Préconditions:

prix(t, p) est défini ssi existe(t, p)
supprimer(t, p) est défini ssi existe(t, p).
ajouter(t, p, px) est défini ssi non existe(t, p).

t est un tarif
p est un produit
px est un prix, un réel.

Axiomes: existe, nbProduits, prix sont les 3 observateurs donc.

existe(nouveau(), p) = faux
existe(ajouter(t, p, px), p) = vrai
existe(ajouter(t, p1, px), p2) = existe(t, p2). si $p1 \neq p2$
existe(supprimer(t, p1), p1) = faux
existe(supprimer(t, p1), p2) = existe(t, p2) si $p1 \neq p2$

nbProduits(nouveau()) = 0
nbProduits(ajouter(t, p, px)) = nbProduits(t) + 1
nbProduits(supprimer(t, p)) = nbProduits(t) - 1

prix(ajouter(t, p, px), p) = px
prix(ajouter(t, p1, px), p2) = prix(t, p2) si $p1 \neq p2$.
prix(supprimer(t, p1), p2) = prix(t, p2) si $p1 \neq p2$.

SDD
5

Chapitre Listes : Spécification, Tests et Implantations.

Structure séquentielle :

$\overset{0}{\square}\ \square\ \square\ \dots\ \overset{n}{\square}$ index n.
ordre total sur les éléments.

- ajouter un élément.
- supprimer un élément.

Type Liste [E].

Opérations :

fonction interne:

empty : → Liste [E] -- créer une liste vide.
addFirst : Liste [E] × E → Liste [E] -- ajouter un élément en tête.
addLast : Liste [E] × E → Liste [E] -- ajouter un élément en queue.
add : Liste [E] × E × Entier → Liste [E] -- ajouter un élément à un rang donné
remove : Liste [E] × Entier → Liste [E] -- supprimer un élé. de rang donné.
set : Liste [E] × E × Entier → Liste [E] -- modifier un élé. de rang donné.
queue : Liste [E] → Liste [E] -- liste privée de sa tête.

Opérateurs:

size : Liste [E] → Entier -- nombre d'éléments
first : Liste [E] → E -- premier élément en tête de liste
isEmpty : Liste [E] → booléen. -- vrai si la liste est vide.
get : Liste [E] × Entier → E -- élé. de rang donné
contains : Liste [E] × E → booleen -- vrai si l'élé ∈ liste.
indexOf : Liste [E] × E → Entier -- rang d'un élé.

Préconditions :

add (l, e, i) est défini ssi $0 \leq i \leq size(l)$
remove (l, i) est défini ssi $0 \leq i < size(l)$
set (l, e, i) est défini ssi $0 \leq i < size(l)$.

queue (l) est défini si non isEmpty (l)
first (l) ———————————————
get (l, i) ——————— $0 \leq i < size(l)$.

Axiomes :

size()

$\begin{cases} size(empty()) = 0 \\ size(addFirst(l, e)) = size(l) + 1 \\ size(addLast(l, e)) = size(l) + 1 \\ size(add(l, e, i)) = size(l) + 1 \\ size(remove(l, i)) = size(l) - 1 \\ size(set(l, e, i)) = size(l) \\ size(queue(l)) = size(l) - 1 \end{cases}$

faire pareil pour "first" et "isEmpty".

$get(add(l, e, i), i) = e$

$get(add(l, e, i), j) = \begin{cases} get(l, j) & si\ j < i \\ get(l, j-1) & si\ j > i \end{cases}$

$get(remove(l, i), j) = \begin{cases} get(l, j) & si\ j < i \\ get(l, j+1) & si\ j \geq i \end{cases}$

$get(set(l, e, i), i) = e$

$get(set(l, e, i), j) = get(l, j)$ si $i \neq j$.

$get(queue(l), i) = get(l, i+1)$.

SDD
6

Tests :

Deux implantations de Listes.
- par tableau
- par liste doublement chaînée

TestListe | classe de Tests

abstraite

TestFirst
testSize
test Get
test ...

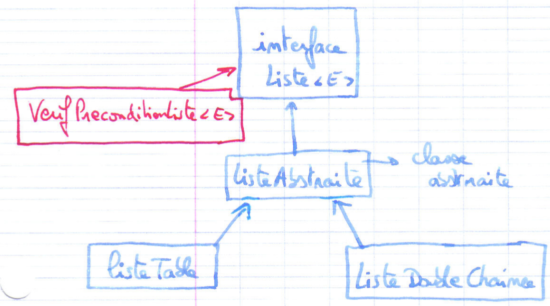
tous les observateurs.

Classe abstraite car les setUp sont abstraits

TestTab
avec définition des setUp

TestListeDoubleChainee
avec définition des setUp.

Exemple : add (ℓ, e, i) défini ssi $0 \leq i < size(\ell)$
spécification (types abstraits) est implanté par une interface Java.
dans l'interface Liste<E> : public void add (E el, int rang) ;



addLast pourrait être écrite complètement
public void addLast (E el) {
this. add (el, this. size)}
les autres sont déclarées abstraites.

Que faire de la précondition du add?

```
assert ( rang >= 0   &&   rang <= size() ) :
        " violation de rang >= 0  &&  rang <= size() " ;
```

<u>Méthode naïve :</u>

écrire les préconditions dans chacune des classes "d'implementation"
listeTable
liste Double Chainee.

<u>Méthode en utilisant le pattern décorateur.</u>

```
public class VerifPreconditionListe <E> implements Liste <E> {
        protected Liste <E> liste ;

        public void add ( E el, int rang) {
          // on écrit l'assert de add.
          assert( ... );
           liste.add (E, rang);
        }
}
```

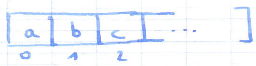
SDD
7

Ecrire dans ListeAbstraite : add, addLast, first, isEmpty.

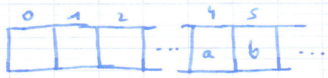
```
package listes;
public abstract class ListeAbstraite <E> implements Liste<E> {
    protected int size;

    public abstract void add (E el, int rang);
    public void addLast (E el) {
        add (el, size);
    }
    public E first() {
        return get (0);
    }
    public boolean isEmpty() {
        return size == 0;
    }
```

$0 \leq rang \leq size - 1.$

1ère solution :

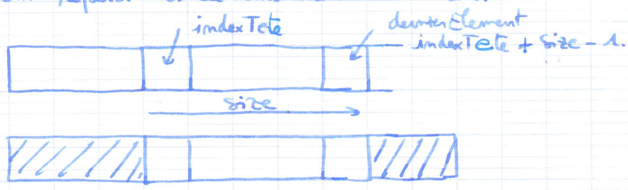


2ème solution
"liste circulaire"

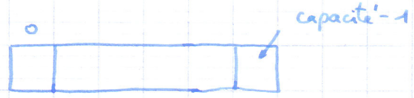


pour les adjonctions en tête et en queue, il n'est pas nécessaire d'effectuer des décalages.

Comment repérer les éléments dans le tableau.

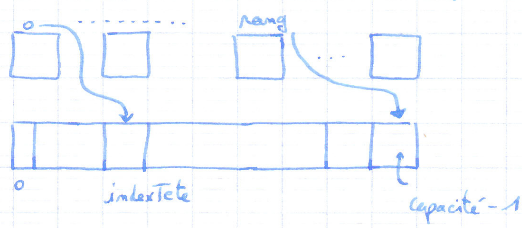


En Java :




```
// attributs
        protected Object [] elements;
        protected int indexTete ;
        protected int capacite ;

// constructeurs
        public listTable (int capacite) {
            assert (capacite > 0) : " capacite > 0" ;
            elements = new Object [capacite] ;
            this. capacite = capacite ;
            size = 0;
        }
```

Calcul de l'indice dans le tableau correspondant au rang dans la liste.




```
private int indCirc (int rang) {
    assert (rang >= 0 && rang <= size) : "rang >= 0 && rang <= size" ;
    return (indexTete + rang) % capacite ;
}
```

SDD
8

```
//  implantation de la méthode add dans la classe ListeTable

public void add(E e, int index) {
    if (size == capacite){ // on redimensionne le tableau
        Object [] t = new Object[2*capacite];
        for (int i = 0; i < size(); i++) {
            t[i] = elements[indCirc(i)];
        };
        capacite = 2 * capacite;
        indexTete = 0;
        elements = t;
    };
    if (index < size()/2) {
        // On décale à gauche
        for (int i = 0; i < index; i++) {
            elements[indCirc(i-1)] = elements[indCirc(i)];
        }
        indexTete = indexTete - 1;
        if (indexTete < 0) indexTete = indexTete+capacite;
    }
    else {
        // on décale à droite
        for (int i = size()-1; i >= index; i--)
            elements[indCirc(i+1)] = elements[indCirc(i)];
    }
    elements[indCirc(index)] = e;
    size = size + 1;
}
```

```
public void remove (int index) {
    if (index < size() / 2) { // décalage à droite.
      for (int i = index; i >= 1 ; i--)
          elements [indCirc (i)] = elements [indCirc(i-1)]
          indTete ++;
          if (indTete > capacite) indTete = 0;
        (// ou indTete = indTete % capacite;)
    }
    else { // décalage à gauche
        for (int i = index ; i < size() - 1 ; i++)
          elements [indCirc(i)] = elements [indCirc(i+1)];
    }
    size -- ;
} //remove.
```

```java
public E get (int i) {
    return (E) elements [indCirc (i)] ;
}

public void set (int index, E e) {
    elements [indCirc (index)] = e ;
}
```

Implantation : Listes doublement chaînées

```java
public class LeftRightNode <T> {
    protected LeftRightNode <T> left ;
    protected LeftRightNode <T> right ;
    protected T valeur ;
}

public class ListeDoublementChainee <E> extends ListeAbstraite <E> {
    protected LeftRightNode <E> sentinelleGauche ;
    ______________________________ sentinelleDroite ;
    ______________________________ curseur ; // pour accélérer les parcours.
    protected int rangCourant ; // rang du curseur.

    public ListeDoublementChainee () {
        super () ; // pour initialiser size.
        sentinelleGauche = new LeftRightNode () ;
        sentinelleDroite = new LeftRightNode () ;

        sentinelleDroite . setLeft (sentinelleGauche) ;
        sentinelleGauche . setRight (sentinelleDroite) ;
        curseur = sentinelleGauche ;
        rang = -1 ;
    }
```