Livret - 7

Implantation de types abstraits

---

liste, pile, file,

lifo générique, fifo générique.

Outil utilisé : classes C#

RM di scala

Cours informatique programmation

Rm di Scala - http://www.discala.net

7 : Types abstraits de données

Implantation avec des classes en C#

Plan du chapitre:

1. Types abstraits de données et classes en C#

Dans cette partie nous adoptons un point de vue pratique dirigé par l'implémentation sous une forme accessible à un débutant des notions de type abstrait de donnée.

Nous allons proposer une écriture des TAD avec des classes C# :

- La notion classique de **classe**, contenue dans tout langage orienté objet, se situe au niveau 2 de cette hiérarchie **constitue une meilleure approche de la notion de TAD**.

En fait un TAD sera bien décrit par les membres **« public »** d'une **classe** et se traduit presque immédiatement ; le travail de traduction des préconditions est à la charge du développeur et se trouvera dans le corps des méthodes.

1.1 Traduction générale TAD → classe C#

Nous proposons un tableau de correspondance pratique entre la signature d'un TAD et les membres d'une classe :

syntaxe du TAD	*squelette de la classe associée*
TAD Truc	**class Truc {**
utilise TAD_1 , TAD_2,...,TAD_n	**Les classes** TAD_1,...,TAD_n sont déclarées dans le même espace de nom que truc (sinon le global par défaut)
champs **.....**	**attributs** **.....**
opérations Op1 : E x F → G Op2 : E x F x G → H x S	**public** G Op1(E x, F y) {} **public void** Op2(E x, F y, G, **ref** H t, **ref** S u) { } **.....**
FinTAD-Truc	**}**

Reste à la charge du programmeur l'écriture du code dans le corps des méthodes Op1 et Op2

1.2 Exemples de Traduction TAD → classe C#

Le TAD Booléens implanté sous deux spécifications concrètes en C# avec deux types scalaires différents.

Spécification opérationnelle concrète n°1

Les constantes du type Vrai, Faux sont représentées par deux attributs de type entier dans un type structure nommé « logique » :

```
public struct logique
{
    static public int Faux = 0;
    static public int Vrai = 1;
}
```

Voici l'interface de la unit traduite et le TAD :

TAD : Booléens
Champs :
Opérations :
Vrai : → Booléens
Faux : → Booléens
Et : Booléens x Booléens → Booléens
Ou : Booléens x Booléens → Booléens
Non : Booléens → Booléens
 	
FINTAD-Booléens
	class Booleens
	{
	public logique val**;**
	
	public Booleens Et (Booleens x, Booleens y) { }
	public Booleens Ou (Booleens x, Booleens y) { }
	public Booleens Non (Booleens x) { }
	
	}

Spécification opérationnelle concrète n°2

Les constantes du type Vrai, Faux sont représentées par deux identificateurs C# dans un type énuméré nommé « logique » :

```
public enum logique { Faux, Vrai };
```

Voici l'interface de la unit traduite et le TAD :

TAD : Booléens
Champs :
Opérations :
Vrai : → Booléens
Faux : → Booléens
Et : Booléens x Booléens → Booléens
Ou : Booléens x Booléens → Booléens
Non : Booléens → Booléens
 	
FINTAD-Booléens
	class Booleens
	{
	public logique val ;
	
	public Booleens Et (Booleens x, Booleens y) { }
	public Booleens Ou (Booleens x, Booleens y) { }
	public Booleens Non (Booleens x) { }
	
	}

Nous remarquons la forte similarité des deux spécifications concrètes :

Implantation avec des entiers	Implantation avec des énumérés

public struct logique
{
static public int Faux = 0;
static public int Vrai = 1;
}
	public enum logique = (faux , vrai) ;

```
class Booleens
{
  public int val ;
  public Booleens (int init)
  {
    val = init;
  }
  public Booleens Et (Booleens x, Booleens y ) { }
  public Booleens Ou (Booleens x, Booleens y ) { }
  public Booleens Non (Booleens x ) { }
}
```

```
class Booleens
{
  public logique val ;
  public Booleens (logique init)
  {
    val = init;
  }
  public Booleens Et (Booleens x, Booleens y ) { }
  public Booleens Ou (Booleens x, Booleens y ) { }
  public Booleens Non (Booleens x ) { }
}
```

1.3 Variations sur les spécifications d'implantation

Cet exercice ayant été proposé à un groupe d'étudiants, nous avons eu plusieurs genres d'implantation des opérations : « et », « ou », « non ». Nous exposons au lecteur ceux qui nous ont parus être les plus significatifs :

Implantation d'après spécification concrète n°1

Fonction Et	Fonction Et

public Booleens Et (Booleens x, Booleens y)
{
  return x.val * y.val ;
}
	public Booleens Et (Booleens x, Booleens y)
	{
	  if (x.val == logique.Faux)
	  return new Booleens (logique.Faux);
	  else return new Booleens (y.val);
	
	}

Fonction Ou	Fonction Ou

public Booleens Ou (Booleens x, Booleens y)
{
  return x.val +y.val - x.val *y.val ;
}
	public Booleens Ou (Booleens x, Booleens y)
	{
	  if (x.val == logique.Vrai)
	  return new Booleens (logique.Vrai);
	  else return new Booleens (y.val);
	}

Fonction Non	Fonction Non

public Booleens Non (Booleens x)
{
  return 1-x.val ;
}
	public Booleens Non (Booleens x)
	{
	  if (x.val == logique.Vrai)
	  return new Booleens (logique.Vrai);
	  else return new Booleens (logique.Faux);
	}

Dans la colonne de gauche de chaque tableau, l'analyse des étudiants a été dirigée par le choix de la spécification concrète sur les entiers et sur un modèle semblable aux fonctions indicatrices des ensembles. Ils ont alors cherché des combinaisons simples d'opérateurs sur les entiers fournissant les valeurs adéquates.

Dans la colonne de droite de chaque tableau, l'analyse des étudiants a été dirigée dans ce cas par des considérations axiomatiques sur une algèbre de Boole. Ils se sont servis des propriétés d'absorbtion des éléments neutres de la loi " *ou* " et de la loi " *et* ". Il s'agit là d'une structure algébrique abstraite.

Influence de l'abstraction sur la réutilisation

A cette étape du travail nous avons demandé aux étudiants quel était, s'il y en avait un, le meilleur choix d'implantation quant à sa réutilisation pour l'implantation d'après la spécification concrète n°2.

Les étudiants ont compris que la version dirigée par les axiomes l'emportait sur la précédente, car sa qualité d'abstraction due à l'utilisation de l'axiomatique a permis de la réutiliser sans aucun changement dans la partie **implementation** de la unit associée à spécification concrète n°2 (en fait toute utilisation des axiomes d'algèbre de Boole produit la même efficacité).

Conclusion :

l'abstraction a permis ici une réutilisation totale et donc un gain de temps de programmation dans le cas où l'on souhaite changer quelle qu'en soit la raison, la spécification concrète.

1.4 Exemples d'implantation de liste linéaire générique

Rappel de l'écriture du TAD de liste linéaire

TAD Liste
utilise : **N**, T_0, Place
Champs : $(a_1,.....,a_n)$ suite finie dans T_0
opérations :
liste_vide : → Liste
acces : Liste x N → Place
contenu : Place → T_0
kème : Liste x **N** → T_0
long : Liste → **N**
supprimer : Liste x **N** → Liste
inserer : Liste x **N** x T_0 → Liste
ajouter : Liste x T_0 → Liste
succ : Place → Place

préconditions :
acces(L,k) **def_ssi** $1 \le k \le long(L)$
supprimer(L,k) **def_ssi** $1 \le k \le long(L)$
inserer(L,k,e) **def_ssi** $1 \le k \le long(L) + 1$
kème(L,k) **def_ssi** $1 \le k \le long(L)$

Fin-Liste

Dans les exemples qui suivent, la notation ≅. indique la traduction en langage C#.

spécification proposée en C# :

Liste ≅	interface IListe<T0> {
	int longueur
	{
	get;
	}
	T0 this[int index]
	{
	get;
	set;
	}
	bool est_Vide();
	void ajouter(T0 Elt);
	void inserer(int k, T0 x);
	void supprimer(int k);
	bool estPresent(T0 x);
	int rechercher(T0 x);
	}
	class Liste<T0> : IListe<T0> {
	public static readonly int max_elt = 100;
	private T0[] t;
	private int longLoc;
	public Liste()
	{
	t = new T0[max_elt];
	longLoc = 0;
	}
	// Le reste implante l'interface IListe<T0> ….…..
	}

liste_vide ≅	**Propriété** : longueur = 0
acces ≅	**Indexeur** : this[index]
contenu ≅	**Indexeur** : this[index]
kème ≅	**Indexeur** : this[index]
long ≅	**Propriété** : longueur
succ ≅	**Indexeur** : this[index]
supprimer ≅	public void supprimer (int k) { ..... }
inserer ≅	public void inserer (int k , T_0 Elt) { ..... }

La précondition de l'opérateur **supprimer** peut être ici implantée par le test :

```
if (0 <= k &&  k <= longueur - 1) …….
```

La précondition de l'opérateur **insérer** peut être ici implantée par le test :

```
if (longueur < max_elt && (0 <= k && k <= longueur) ) …….
```

Les deux objets **acces** et **contenu** ne seront pas utilisés en pratique, car l'indexeur de la classe les implante automatiquement d'une manière transparente pour le programmeur.

Le reste du programme est laissé au soin du lecteur qui pourra ainsi se construire à titre didactique, sur sa machine, une base de types en C# de base, il en trouvera une correction à la fin de ce chapitre.

Nous pouvons " **enrichir** " le TAD Liste en lui adjoignant deux opérateurs test et rechercher (rechercher un élément dans une liste). Ces adjonctions ne posent aucun problème. Il suffit pour cela de rajouter au TAD les lignes correspondantes :

opérations
estPresent : Liste x T_0 → Booléen
rechercher : Liste x T_0 → Place
précondition
rechercher(**L**,**e**) **def_ssi** Test(**L**,**e**) = **V**

Le lecteur construira à titre d'exercice l'implantation C# de ces deux nouveaux opérateurs en étendant le programme déjà construit. Il pourra par exemple se baser sur le schéma de représentation C# suivant :

```
public bool estPresent(T0 Elt)
{
 return !(rechercher(Elt) == -1);
}

public int rechercher(T0 Elt)
{
    // il s'agit de fournir le rang de x dans la liste
    // utiliser par exemple un algo de recherche séquentielle
}
```

1.5 Exemples d'implantation de pile LIFO générique

Rappel de l'écriture du TAD Pile LIFO

TAD PILIFO

utilise : T_0, Booléens
Champs : $(a_1,.....,a_n)$ suite finie dans T_0
opérations :
sommet : → PILIFO
Est_vide : PILIFO → Booléens
empiler : PILIFO x T_0 x sommet → PILIFO x sommet
dépiler : PILIFO x sommet → PILIFO x sommet x T_0
premier : PILIFO → T_0
préconditions :
dépiler(P) **def_ssi** est_vide(P) = **Faux**
premier(P) **def_ssi** est_vide(P) = **Faux**

FinTAD-PILIFO

Nous allons utiliser un **tableau** avec une case supplémentaire permettant d'indiquer que le fond de pile est atteint (la case 0 par exemple, qui ne contiendra jamais d'élément).

spécification proposée en C# :

Pilifo ≅	interface ILifo<T0>
	{
	int nbrElt
	{
	get;
	}
	bool estVide();
	void empiler(T0 Elt);
	T0 depiler();
	T0 premier();
	}
	
	class Lifo<T0> : ILifo<T0>
	{
	public static readonly int max_elt = 100;
	private T0[] t;
	private int sommet, fond;
	public Lifo()
	{
	t = new T0[max_elt];
	fond = -1;
	sommet = fond;
	// Le reste implante l'interface ILifo<T0> ….…..
	}

depiler ≅	public T0 depiler ( )
empiler ≅	public T0 empiler (T_0 Elt)
premier ≅	public T0 premier ( )
Est_vide ≅	public bool estVide ( )

Le contenu des méthodes est conseillé au lecteur à titre d'exercice, il en trouvera une correction à la fin de ce chapitre..

Remarque :

Il est aussi possible de construire une spécification opérationnelle à l'aide du TAD Liste en remplaçant dans l'étude précédente le mot " tableau " par le mot " liste ". Il est vivement conseillé au lecteur d'écrire cet exercice en C# pour bien se convaincre de la différence entre les niveaux d'abstractions.

1.6 Exemples d'implantation de file FIFO générique

Rappel de l'écriture du TAD file FIFO

TAD FIFO
utilise : T_0, Booléens
Champs : $(a_1,.....,a_n)$ suite finie dans T_0
opérations :
tête : → FIFO
fin : → FIFO
Est_vide : FIFO → Booléens
ajouter : FIFO x T_0 x fin → PILIFO x fin
retirer : FIFO x tête → FIFO x tête x T_0
premier : FIFO → T_0
préconditions :
retirer(F) **def_ssi** est_vide(F) = **Faux**
premier(F) **def_ssi** est_vide(F) = **Faux**

FinTAD-FIFO

Nous allons utiliser ici aussi un **tableau** avec une case supplémentaire permettant d'indiquer que la file est vide (la case 0 par exemple, qui ne contiendra jamais d'élément).

spécification proposée en C# :

Fifo ≅

```
interface IFifo<T0> {
    int nbrElt
        {  get;   }
    bool estVide();
    void ajouter(T0 Elt);
    T0 retirer();
    T0 premier();
 }

 class Fifo<T0> : IFifo<T0> {
    public static readonly int max_elt = 100;
    private T0[] t;
      private int tete, fin;
    public Fifo()
    {
       t = new T0[max_elt];
       tete = -1;
       fin = 0;
```

	} }
retirer ≅	public T0 retirer ( )
ajouter ≅	public T0 ajouter (T_0 Elt)
premier ≅	public T0 premier ( )
Est_vide ≅	public bool estVide ( )

Le contenu des méthodes est conseillé au lecteur à titre d'exercice, il en trouvera une correction à la fin de ce chapitre..

Remarque :

Comme pour le TAD Pilifo, il est aussi possible de construire une spécification opérationnelle du TAD FIFO à l'aide du TAD Liste en remplaçant dans l'étude précédente l'objet de tableau « private T0[] t » par l'objet de liste « private Liste<T_0> t » .

Une solution d'implantation de liste linéaire générique en C

```
using System;
using System.Collections.Generic;
using System.Text;

namespace cci
{
 interface IListe<T0>
  {
     int longueur
     {
        get;
     }
     T0 this[int index]
     {
        get;
        set;
     }
     bool est_Vide();
     void ajouter(T0 Elt);
     void inserer(int k, T0 x);
     void supprimer(int k);
     bool estPresent(T0 x);
     int rechercher(T0 x);
  }
  class Liste<T0> : IListe<T0>
  {
     public static readonly int max_elt = 100;
     private T0[] t;
     private int longLoc;
     public Liste()
     {
        t = new T0[max_elt];
        longLoc = 0;
     }

     public void ajouter(T0 Elt)
     {
        if (longueur < max_elt)
        {
           t[longueur] = Elt;
           longueur++;
        }
        else
           System.Console.WriteLine("ajout impossible : capacité maximale atteinte !");
     }

     public bool est_Vide()
     {
        return longueur == 0;
     }

     public T0 this[int index]
     {
        get
        {
```

```
            if (!est_Vide())
            {
               if (0 <= index && index <= longueur)
               {
                  return t[index];
               }
               else
               {
                  System.Console.WriteLine("indexage, lecture incorrecte : indice (" + index + ") hors de la liste.");
                  return default(T0);
               }
            }
            else
            {
               System.Console.WriteLine("indexage, lecture incorrecte : Désolé la liste est vide.");
               return default(T0);
            }
         }
         set {
            if (0 <= index && index <= longueur - 1)
            {
               t[index] = value;
            }
            else
               System.Console.WriteLine("indexage, écriture impossible : indice (" + index + ") hors de la liste.");
          }
      }

      public int longueur
      {
         get { return longLoc; }
         protected set { longLoc = value; }
      }

      public void supprimer(int k)
      {
         if (!est_Vide())
         {
            if (0 <= k && k <= longueur - 1)
            {
               for (int i = k; i < longueur - 1; i++)
                  t[i] = t[i + 1];
               longueur--;
            }
            else
               System.Console.WriteLine("suppression impossible : indice (" + k + ") hors de la liste.");
         }
      }

      public void inserer(int k, T0 Elt)
      {
         if (!est_Vide())
         {
            if (longueur < max_elt && (0 <= k && k <= longueur))
            {
               for (int i = longueur - 1; i >= k; i--)
                  t[i + 1] = t[i];
               t[k] = Elt;
               longueur++;
            }
```

```
            else
               if (longueur >= max_elt)
                  System.Console.WriteLine("insertion impossible : capacité maximale atteinte !");
               else
                  System.Console.WriteLine("insertion impossible : indice (" + k + ") hors de la liste.");
         }
         else
            ajouter(Elt);
      }

      public bool estPresent(T0 Elt)
      {
         return !(rechercher(Elt) == -1);
      }

      public int rechercher(T0 Elt)
      {
         int k;
         for (k = 0; k < longueur; k++)
            if (Elt.Equals(t[k])) break;
         if (k == longueur)
            return -1;
         else
            return k;
      }

      public void afficher()
      {
         for (int k = 0; k < longueur; k++)
            System.Console.Write(t[k] + ", ");
         System.Console.WriteLine(": long = " + longueur);
      }
   }
}
```

Exemple de classe principale testant la classe « Liste » sur T_0 = **int** :

```
class ProgramListe
{
   static void Main(string[] args)
   {
      Liste<int> liste = new Liste<int>();
      System.Console.WriteLine(liste[0]);
      for (int i = 0; i < Liste<int>.max_elt / 10; i++)
         liste.inserer(i, i);
      liste.afficher();
      liste.inserer(20, 97);
      liste.inserer(liste.longueur, 98);
      liste.supprimer(liste.longueur - 1);
      liste.afficher();
      ……
   }
}
```

Une solution d'implantation de pile Lifo générique en C#

```
interface ILifo<T0>
{
   int nbrElt
   {
      get;
   }
   bool estVide();
   void empiler(T0 Elt);
   T0 depiler();
   T0 premier();
}

class Lifo<T0> : ILifo<T0>
{
   public static readonly int max_elt = 100;
   private T0[] t;
   private int sommet, fond;
   public Lifo()
   {
      t = new T0[max_elt];
      fond = -1;
      sommet = fond;
   }
   public int nbrElt
   {
      get { return sommet + 1; }
   }

   public bool estVide()
   {
      return sommet == fond;
   }

   public void empiler(T0 Elt)
   {
      if (sommet < max_elt)
      {
         sommet++;
         t[sommet] = Elt;
      }
      else
         System.Console.WriteLine("empilement impossible : capacité maximale atteinte !");
   }

   public T0 depiler()
   {
      if (!estVide())
      {
         T0 Elt = t[sommet];
         sommet--;
         return Elt;
      }
      else
      {
```

```
            System.Console.WriteLine("dépilement impossible : pile vide !");
            return default(T0);
        }
    }

    public T0 premier()
    {
        if (!estVide())
        {
            return t[sommet];
        }
        else
        {
            System.Console.WriteLine("premier impossible : pile vide !");
            return default(T0);
        }
    }

    public void afficher()
    {
        for (int k = 0; k <= sommet; k++)
            System.Console.Write(t[k] + ", ");
        System.Console.WriteLine(": nbr elmt = " + this.nbrElt);
    }
}
```

Exemple de classe principale testant la classe « Lifo » sur T_0 = **int** :

```
class ProgramListe
{
    static void Main(string[] args)
    {
            Lifo<int> pile = new Lifo<int>();
            pile.afficher();
            pile.depiler();
            System.Console.WriteLine("premier=" + pile.premier());
            for (int i = 0; i < Lifo<int>.max_elt / 10; i++)
              pile.empiler(i);
            pile.afficher();
            System.Console.WriteLine("on dépile : " + pile.depiler());
            System.Console.WriteLine("on dépile : " + pile.depiler());
            pile.afficher();
    }
}
```

Une solution d'implantation de file Fifo générique en C#

```
using System;
using System.Collections.Generic;
using System.Text;

   interface IFifo<T0>
   {
      int nbrElt
      {
         get;
      }
      bool estVide();
      void ajouter(T0 Elt);
      T0 retirer();
      T0 premier();
   }

   class Fifo<T0> : IFifo<T0>
   {
      public static readonly int max_elt = 100;
      private T0[] t;
      private int tete, fin;

      private void decaleUn()
      {
         if (tete < max_elt)
         {
            tete++;
            for (int k = tete; k >= 0; k--)
               t[k + 1] = t[k];
         }
      }
      public Fifo()
      {
         t = new T0[max_elt];
         tete = -1;
         fin = 0;
      }
      public int nbrElt
      {
         get { return tete + 1; }
      }

      public bool estVide()
      {
         return tete == -1;
      }

      public void ajouter(T0 Elt)
      {
         if (tete < max_elt)
         {
            decaleUn();
            t[fin] = Elt;
         }
```

```
        else
            System.Console.WriteLine("ajout impossible : capacité maximale atteinte !");
    }

    public T0 retirer()
    {
        if (!estVide())
        {
            T0 Elt = t[tete];
            tete--;
            return Elt;
        }
        else
        {
            System.Console.WriteLine("ajout impossible : file vide !");
            return default(T0);
        }
    }

    public T0 premier()
    {
        if (!estVide())
        {
            return t[tete];
        }
        else
        {
            System.Console.WriteLine("premier impossible : file vide !");
            return default(T0);
        }
    }

    public void afficher()
    {
        for (int k = fin; k <= tete; k++)
            System.Console.Write(t[k] + ", ");
        System.Console.WriteLine(": nbr elmt = " + this.nbrElt);
    }
}
class ProgramFifo
{
    static void Main(string[] args)
    {
        Fifo<int> file = new Fifo<int>();
        file.afficher();
        file.retirer();
        System.Console.WriteLine("premier = " + file.premier());
        for (int i = 0; i < Fifo<int>.max_elt / 10; i++)
            file.ajouter(i);
        file.afficher();
        System.Console.WriteLine("on retire : " + file.retirer());
        System.Console.WriteLine("on retire : " + file.retirer());
        file.afficher();
    }
}
```

Exemple de classe principale testant la classe « Fifo » sur T_0 = **int** :

```
class ProgramFifo
{
    static void Main(string[] args)
    {
        Fifo<int> file = new Fifo<int>();
        file.afficher();
        file.retirer();
        System.Console.WriteLine("premier = " + file.premier());
        for (int i = 0; i < Fifo<int>.max_elt / 10; i++)
            file.ajouter(i);
        file.afficher();
        System.Console.WriteLine("on retire : " + file.retirer());
        System.Console.WriteLine("on retire : " + file.retirer());
        file.afficher();
    }
}
```

Résultats d'exécution sur la console :

: nbr elmt = 0
ajout impossible : file vide !
premier impossible : file vide !
premier = 0
9, 8, 7, 6, 5, 4, 3, 2, 1, 0, : nbr elmt = 10
on retire : 0
on retire : 1
9, 8, 7, 6, 5, 4, 3, 2, : nbr elmt = 8