Rapport de projet

Par Ophélie Cluzel

Partie 1 : 1ère application POO (Gestionnaire des objets du jeu Fortnite)

I. Cahier des charges

Le but de ce projet est de créer une application qui soit un gestionnaire d'équipements issus du jeu Fortnite Battle Royale. Ces équipements sont constitués des tenues du joueur, appelés Skins, et d'autres objets qui sont des Pioches et des Planneurs. Ces équipements possèdent chacun un nom, une description, une rareté, un prix en V-Bucks (qui est la monnaie virtuelle du jeu), une date de parution dans le jeu et bien sur une représentation sous forme d'image.

Pour ce faire, il va falloir créer différentes classes objets et leurs classes collection d'objets associées ainsi qu'un gestionnaire qui sera intitulé ManagerFBR.

II. Analyse fonctionnelle

Passons maintenant à l'analyse concrète de ce que contient mon application.

Toutes les classes que je vais décrire ci-dessous sont situées dans un même espace de nom qui porte le nom du projet :

```
namespace ProjetFBR
```

C'est important que toutes les classes du projet soient dans un même espace de nom car elles pourront ainsi s'appeler entre elles.

Tout d'abord il va falloir créer nos différents équipements, comme on constate que tous les équipements ont les mêmes attributs, on peut donc créer une classe parent « Equipement » dont vont hériter les classes Skin, Pioche et Planneur.

❖ Analyse de la classe Equipement :

```
abstract class Equipement : IEquatable<Equipement>, IComparable<Equipement>
```

Comme on peut le voir ci-dessus, la classe Equipement implémente deux interfaces : l'interface IEquatable<> et l'interface IComparable<>.

1. `IEquatable<Equipement>`

L'interface IEquatable<> permet de créer une méthode spécifique à notre classe Equipement permettant de déterminer l'égalité des instances. Pour l'utiliser, il faut implémenter la méthode Equals() de l'interface et également surcharger les opérateurs d'égalité « == » et « != ».

Or, on peut constater que dans le programme il y a deux méthodes Equals() :

```
public bool Equals(Equipement other)
```

La première qui va prendre en paramètre un objet de type Equipement est simplement la méthode Equals() de la classe Equipement. Comme on compare les Equipements en fonction de leur noms, qui est un attribut de type string, la méthode va faire appel au Equals() de la classe string.

```
public override bool Equals(object obj)
```

Enfin la deuxième qui va prendre en paramètre un objet de type object est en fait cette fois-ci l'implémentation de l'interface IEquatable<> et celle-ci va faire appel à la fonction Equals() de la classe Equipement.

Ensuite on surcharge les opérateurs « == » et « != » grâce aux méthodes Equals() que l'on vient de définir. Si on en surcharge un, il faut impérativement surcharger l'autre.

```
public static bool operator ==(Equipement e1, Equipement e2)

public static bool operator !=(Equipement e1, Equipement e2)
```

On peut voir ici que les méthodes sont déclarées « static », cela signifie qu'elles peuvent s'exécuter sans avoir à instancier la classe qui les contient.

Pour finir il faut surcharger la méthode GetHashCode() ce qui peut être utile pour les collections nécessitants du hachage. Ici on va définir le HashCode sur l'attribut Nom (car 2 équipements ne peuvent pas avoir le même nom).

```
public override int GetHashCode()
```

2. IComparable<Equipement>

L'interface IComparable<> permet de définir une méthode de comparaison propre à la classe Equipement et lui permettant d'ordonnancer ou de trier ses instances. Pour l'utiliser, il faut implémenter la méthode CompareTo() et également surcharger les opérateurs de comparaison « > », « < », « >= » et « <= » .

```
public int CompareTo(Equipement other)
```

Comme on veut comparer les équipements en fonction de leurs noms, et que l'attribut nom est de type string, la méthode CompareTo() va faire appel à la méthode Compare() de la classe string pour comparer le nom de l'équipement courant avec celui de l'équipement passé en paramètre.

```
public static bool operator >=(Equipement e1, Equipement e2)

public static bool operator <=(Equipement e1, Equipement e2)

public static bool operator >(Equipement e1, Equipement e2)

public static bool operator <(Equipement e1, Equipement e2)
```

Les méthodes de surcharges des opérateurs de comparaison vont donc faire appel à la méthode CompareTo() qui vient d'être définie. Ces méthodes sont déclarées « static » pour la même raison que les surcharges d'opérateur d'égalité vues précédemment. Si on surcharge l'opérateur « >= » on doit impérativement surcharger l'opérateur « <= » et de la même façon si on surcharge l'opérateur « >» on doit impérativement surcharger l'opérateur « <».

Ensuite la classe dispose de différentes propriétés en lecture et en écriture sur ses attributs. Généralement, les propriétés portent le même nom que leur attribut associé mais avec une majuscule.

```
public categorieRarete Rarete

public string Description

public string Nom

public DateTime DateParution

public int Prix

public Image Photo
```

On peut aussi créer des propriétés un peu plus complexes comme par exemple la propriété suivante :

```
public string RareteS
{
    get { return Enum.Format(typeof(categorieRarete), this.rarete, "g"); }
    set { this.rarete = (categorieRarete)Enum.Parse(typeof(categorieRarete), value, false); }
}
```

Elle permet de gérer l'attribut Rareté de l'équipement qui est à la base de type énuméré (Enum) mais sous forme de chaîne de caractères (string). Pour ce faire il faut faire appel aux fonctions Format() et Parse() de la classe Enum pour réaliser le transtypage.

Pour finir notre classe contient une surcharge de la classe ToString() de la classe object et la méthode Affiche() qui va lui faire appel pour afficher notre équipement dans la console.

```
public override string ToString()

public void Affiche()
```

La classe Equipement est déclarée abstract car elle sert uniquement de classe de base pour d'autres classes, et ne sera jamais instanciée toute seule. Ainsi, elle ne dispose pas de constructeur. Les membres définis comme abstraits doivent être implémentés par des classes non abstraites dérivées de la classe abstraite. Ici la méthode Saisie() devra être surchargée dans les sous classes Skin, Pioche et Planneur.

```
public abstract void Saisie();
```

- ❖ <u>Analyse de la classe Skin :</u>

```
class Skin : Equipement
```

La classe Skin ne contient pas grand-chose puisqu'elle va hériter des différentes méthodes et attributs prédéfinis dans la classe parent Equipement.

Elle contient donc un constructeur par défaut et un constructeur avec des paramètres qui vont initialiser les différents attributs via les propriétés qui leur sont associées.

```
public Skin()
```

```
public Skin(string n, string d, int p, categorieRarete cat)
```

Et bien entendu, elle contient la surcharge de la méthode abstraite Saisie() vue précédemment de la classe Equipement dont elle hérite (le mot clef « override » indique que l'on fait une surcharge) qui permet à l'utilisateur de saisir une nouvelle tenue via la console.

```
public override void Saisie()
```

- ❖ Analyse des classes Pioche et Planneur

```
class Pioche : Equipement
```

```
class Planneur : Equipement
```

Les classes Pioche et Planneur sont construites exactement sur le même principe que la classe Skin. On peut voir ici aussi l'héritage de la classe Equipement.

- ❖ Analyse de la classe SkinEvolutif

```
class SkinEvolutif: Skin
```

La classe SkinEvolutif hérite de la classe Skin (et donc par transitivité également de la classe Equipement) et de ses méthodes.

```
private List<string> lNoms;
private List<Image> lPhotos;
```

Un skin évolutif va disposer des mêmes attributs qu'un skin basique mais avec en plus une liste de noms et une liste de photos qui contiendront les noms et photos de ses différentes évolutions.

```
public SkinEvolutif()
```

```
public SkinEvolutif(string n, string d, int p, categorieRarete cat)
```

Cette classe contient donc un constructeur par défaut et un constructeur avec paramètres qui vont initialiser ses différents attributs.

```
public List<Image> LPhotos
```

```
public List<string> LNoms
```

Elle contient également deux propriétés en lecture seule sur ses attributs de type list<>.

```
public override string ToString()
```

Ensuite on surcharge la méthode ToString() héritée de la classe Skin. Comme on a surchargé la méthode ToString() pas besoin de surcharger la méthode Affiche(). En effet, la méthode Affiche() de la classe Skin fera elle-même appel à la méthode ToString() de la classe SkinEvolutif lorsqu'elle aura affaire à un skin de ce type.

```
public override void Saisie()
```

Enfin on surcharge la méthode Saisie() héritée de la classe Skin. Pour cela on commence par récupérer la méthode Saisie() de la classe skin grâce à base.Saisie() puis on rajoute à la suite ce qu'on veut saisir dans le cas d'un skin évolutif (saisir la liste des noms d'évolutions).

```
public void AjoutEvolution(string n,Image p)
{
    LNoms.Add(n);
    LPhotos.Add(p);
}
```

Pour finir on crée une méthode AjoutEvolution() qui va nous permettre d'ajouter une évolution à notre skin évolutif c'est-à-dire de rajouter un nom et une photo dans nos listes de noms d'évolutions et de photos d'évolutions. Pour cela on fait appel à la méthode Add() de la classe List<> qui ajoute simplement un élément à la suite de la liste.

- ❖ Analyse de la classe LesSkins

```
class LesSkins
```

La classe LesSkins va nous permettre de gérer une collection d'objets de type Skin. Pour cela on définit un attribut de type List<Skin>.

Elle dispose donc d'un constructeur par défaut qui va initialiser l'attribut liste.

```
public LesSkins()
```

Elle dispose également de différentes propriétés :

```
public List<Skin> Liste
{
    get { return this.liste; }
}
```

Une propriété sur son attribut liste mais uniquement en lecture, ce qui signifie qu'elle ne pourra pas subir d'affectations extérieures.

```
public int NbSkins
{
    get { return this.liste.Count; }
}
```

Et également une propriété NbSkins qui va comme son nom l'indique renvoyer le nombre de Skins présent dans notre liste. Pour ce faire, elle va faire appel à la propriété Count de la classe List<> qui renvoie le nombre d'éléments de la liste.

```
public void Ajoute(Skin s)

public void Supprime(string nom)
```

La classe LesSkins comporte ensuite une méthode Ajoute() et une méthode Supprime() qui permettent respectivement d'ajouter ou de supprimer un des Skins de la liste. La méthode Ajoute() va faire appel à la méthode Add() de la classe List<>. La méthode Supprime() va supprimer le skin dont on passe le nom en paramètre. Elle va faire appel à la méthode GetByNom() que nous allons voir plus tard qui permet de retrouver un skin dans la liste en fonction de son nom, puis à la méthode Remove() de la classe List<>.

- ❖ Les fonctions de recherche

La classe les Skins contient différentes méthodes de recherche pour retrouver les skins selon une certaine particularité.

```
public Skin GetByNom(string n)
```

Cette méthode va parcourir la liste des skins et renvoyer le skin dont le nom correspond au nom passé en paramètre. Cette méthode renvoie un seul Skin car il ne peut pas y avoir plusieurs Skins de même nom.

```
public LesSkins GetByRarete(categorieRarete cat)
```

Cette méthode va renvoyer tous les Skins correspondants à la rareté passée en paramètre. Comme plusieurs skins peuvent avoir la même rareté on renvoie une valeur de type LesSkins. Pour cela elle commence par créer une nouvelle collection de skins LesSkins, parcourt la liste totale des skins et fait appel aux opérateurs d'égalité qui vérifient si la valeur de rareté de chaque skin est égale à celle passée en paramètre et si oui on l'ajoute à notre nouvelle collection de skins grâce à la fonction Ajoute() de la classe LesSkins.

```
public LesSkins GetByPrix(int p)
```

Même principe que la méthode précédente, on renvoie la liste des skins correspondant au prix passé en paramètre.

```
public LesSkins RechSkinsEvolutifs()
{
    LesSkins ls = new LesSkins();
    foreach (Skin sk in Liste)
    {
        if (sk.GetType().Equals(typeof(SkinEvolutif)))
            ls.Ajoute(sk);
    }
    return ls;
}
```

Enfin pour finir une fonction de recherche des skins évolutifs qui va nous renvoyer la liste des skins de type SkinEvolutif. On commence par parcourir la liste de skins, on vérifie si c'est un skin de type skinEvolutif en faisant appel à la méthode GetType() de la classe object et à la méthode Equals() de la classe Type et si oui, on l'ajoute à la liste grâce à la méthode Ajoute() de la classe LesSkins. Pour finir on renvoie la liste

La classe LesSkins contient également une méthode Exist() qui vérifie si un Skin passé en paramètre fait partie de la collection : si oui elle renvoie un booléen true, sinon false.

```
public Boolean Exist(Skin s)
```

Cette méthode fait appel à la méthode Contains() de la classe List<>.

Nous avons également créé une méthode de tri des skins selon leur nom par ordre alphabétique.

```
public LesSkins TriAlphabetique()
{
    LesSkins lesSkins2 = new LesSkins();
    lesSkins2 = this;
    Skin skintemp = new Skin();
    int compt = lesSkins2.NbSkins;
    do
    {
        for (int i = 1; i < lesSkins2.NbSkins; i++)
        {
            if ((lesSkins2.Liste[i - 1] >= lesSkins2.Liste[i]) == true)
            {
                skintemp = lesSkins2.Liste[i];
                lesSkins2.Liste[i] = lesSkins2.Liste[i - 1];
                lesSkins2.Liste[i - 1] = skintemp;
            }
        }
        compt--;
    } while (compt != 0);
    return lesSkins2;
}
```

On commence par créer une nouvelle collection de skins LesSkins. On lui affecte la valeur de la collection courante. Puis on parcourt la liste des skins et pour on compare les valeurs 2 à 2 :

pour cela on fait appel aux opérateurs de comparaison que l'on a surchargé (qui font eux-mêmes appel à la méthode compareTo() de la classe Skin dérivée de la classe Equipement) qui nous permet de comparer des skins selon leur nom. Si le nom du premier skin est supérieur ou égal dans l'ordre alphabétique au nom du deuxième skin alors on les échange de place dans la liste. Pour cela on crée une variable temporaire pour nous permettre de faire l'échange. On recommence l'opération n fois, n étant le nombre d'éléments dans la liste, pour être sûrs que la liste est bien triée.

Cet algorithme de tri n'est pas très rapide, si on voulait améliorer les performances de l'application il aurait été possible d'utiliser un algorithme de tri déjà existant comme un tri rapide ou un tri fusion.

Ensuite cette classe contient une méthode initAleatoire() et une méthode init(). Ces fonctions vont permettre d'initialiser la liste de skins, respectivement avec des Skins créés avec des valeurs aléatoires, ou des Skins plus concrets basés directement sur les skins tirés du jeu.

```
public void init() { }

public void InitAleatoire(int n)
```

La fonction InitAleatoire() prends en paramètre un entier n qui représente le nombre de skins aléatoires que l'on veut générer et ajouter à la liste.

Pour finir la classe LesSkins contient la surcharge de la méthode toString() de la classe object et la méthode affiche qui lui est associée.

```
public override string ToString()

public void Affiche()
```

La méthode ToString() de la classe LesSkins fait appel à la méthode ToString() de la classe Skin.

- ❖ Analyse des classes LesPlanneurs et LesPioches

Les classes LesPlanneurs et LesPioches sont construites exactement sur le même principe que la classe LesSkins mais avec en attributs respectivement une liste de planneurs et une liste de pioches et toutes les méthodes vues dans LesSkins adaptées aux objets Planneur et Pioche.

```
class LesPlanneurs

class LesPioches
```

- ❖ Analyse de la classe gestionnaire ManagerFBR

La classe ManagerFBR contient des attributs nom, mail et les différentes collections d'équipements vues précédemment.

```
class ManagerFBR
{
    private string nom;
    private string mail;
    private LesSkins ls;
    private LesPioches lpi;
    private LesPlanneurs lpa;
```

Elle contient ensuite les différentes propriétés en lecture et en écriture associées à ces attributs.

```
public string Nom

public string Mail

public LesSkins Ls

public LesPioches Lpi

public LesPlanneurs Lpa
```

Puis évidemment un constructeur par défaut qui va initialiser ses différents attributs.

```
public ManagerFBR()
{
    this.nom = "";
    this.mail = "";
    this.ls = new LesSkins();
    this.lpa = new LesPlanneurs();
    this.lpi = new LesPioches();
}
```

Ensuite il contient la surcharge de la méthode ToString() de la classe object. Cette méthode va faire appel aux différentes méthodes ToString() des classes LesSkins, LesPioches et les Planneurs.

```
public override string ToString()
{
    string s = "";
    s += "\n Nom : " + this.nom;
    s += "\n Mail : " + this.mail;
    s += "\n \n liste des skins: " +this.ls.ToString();
    s += "\n \n liste des pioches: " + this.lpi.ToString();
    s += "\n \n liste des planneurs: " + this.lpa.ToString();
    return s;
}

public void Affiche()
```

La méthode Affiche() va appeler la méthode ToString() définie dans la classe ManagerFBR.

Enfin la classe contient des méthodes de recherche qui vont faire appel à toutes les méthodes de recherche, de tri, d'ajout et de suppression des classes LesSkins, LesPlanneurs et LesPioches.

III. Phase de tests

(Voir Program.cs)

IV. Annexes

Sites internet sources utilisés : https://docs.microsoft.com

https://progameguides.com/fortnite-skins-list/

https://progameguides.com/fortnite-pickaxes-list/

https://progameguides.com/fortnite-gliders-list/

Partie 2 : 2ème application Généricité (Panier d'objets)

I. Cahier des charges

Le but de cette seconde application est de tester le fonctionnement des classes génériques vues en cours de manière théorique. Il est donc demandé de créer une classe Panier<T> où T pourra être remplacé par le type d'objet de notre choix : ici nous allons réutiliser notre classe Skin définie dans la première application.

II. Analyse Fonctionnelle

On commence par déclarer la classe générique Panier<T>.

```
class Panier<T> where T: IEquatable<T>,IComparable<T>
```

On remarque que la déclaration n'est pas seule, en effet l'ajout de where T va nous permettre de préciser que l'objet qui instanciera T devra impérativement avoir implémenté les interfaces IEquatable<T> et IComparable<T>, comme cela on peut être sûr que les méthodes d'égalité (Equals()) et de comparaison (CompareTo()) auront bien été implémentées dans la classe de cet objet : on pourra donc y faire appel dans la classe générique sans que cela ne pose un problème.

```
private T[] tab;
private const int MAX= 6;
private int nb;
```

On déclare ensuite ses différents attributs : tout d'abord un tableau d'éléments T, puis une variable entier constante MAX qui fixe le nombre maximum d'éléments qui pourront être dans ce tableau et enfin pour finir un entier nb qui représentera le nombre courant d'éléments contenus dans le tableau.

La classe contient ensuite différentes méthodes pour la gestion du panier :

- Méthode Ajout

```
public void Ajout(T element)
{   if (this.nb < MAX)
    {
        if (!Appartient(element))
        {
            this.tab[nb++] = element;
        }
        else
            Console.WriteLine("L'élément est déjà dans le panier");
    }
    else
        Console.WriteLine("Le panier est plein, l'élément n'a pas été ajouté");
}
```

Cette méthode permet d'ajouter un élément dans le panier, seulement pour que l'élément puisse y être ajouté il faut respecter plusieurs conditions. Si le nombre d'éléments courants nb est supérieur ou égal à la constante MAX que l'on a fixé alors la méthode renvoie un message d'erreur notifiant que le panier est plein et l'élément n'est pas ajouté. Ensuite, si l'élément appartient déjà au panier, la méthode renvoie un message d'erreur notifiant que l'élément fait déjà partie du panier et il n'est pas rajouté une seconde fois : ainsi on évite les doublons (cela fait appel à la méthode Appartient() que je vais détailler plus tard). Pour finir si le nombre d'éléments courant est bien inférieur au MAX et que l'élément n'appartient pas déjà au panier alors on peut l'ajouter au panier sans oublier d'incrémenter notre variable nb.

- Méthode Supprime

```
public void Supprime(T element)
{
    int indice = RechIndice(element);
    if (indice == -1)
        Console.WriteLine("Cet élément ne fait pas partie du Panier");
    else
    {

        for (int i = indice + 1; i < nb; i++)
            tab[i - 1] = tab[i];
        nb--;
    }
}
```

Cette méthode permet de supprimer un élément dans le panier, seulement pour cela il va falloir vérifier que l'élément est bel est bien dans le panier. Pour cela on utilise la méthode RechIndice() que je vais détailler plus tard, qui renvoie l'indice de l'élément dans le tableau tab si elle l'a trouvé ou -1 s'il n'est pas dans le tableau. Ainsi si l'indice est égal à -1, la méthode Supprime() renvoie un message d'erreur notifiant que l'élément à supprimer ne fait pas partie du panier. Sinon, la méthode va recopier les valeurs du tableau un cran en arrière à partir de l'indice+1, en écrasant ainsi la valeur de l'indice à supprimer. Enfin, il ne faut pas oublier de décrémenter notre variable nb représentant le nombre d'éléments courant .

- Méthode RechIndice

```
public int RechIndice(T element)
{
    int compt = 0;
    while (compt < this.nb)
    {
        if (this.tab[compt].Equals(element))
            return compt;
        else
            compt++;
    }
    return -1;
}
```

Cette méthode permet de renvoyer l'indice dans le tableau d'un élément donné ou -1 s'il n'est pas dans le tableau. Ainsi, la méthode parcourt le tableau et pour chaque indice elle teste si la valeur qu'il contient est égale à l'élément passé en paramètre : cela va faire appel à la méthode Equals() de la classe l'élément. Il était donc important pour cette méthode d'imposer que l'élément devait avoir implémenté l'interface IEquatable<> dans sa classe. Si à la fin du parcours du tableau la méthode n'a rien renvoyé, cela signifie qu'elle n'a pas trouvé l'élément donc elle renvoie -1.

- Méthode Appartient

```
public bool Appartient(T element)
{
    bool b = false;
    int compt = 0;

    if (this.nb == 0)
        return b;
    while (compt < nb && !b)
    {
        if (this.tab[compt].Equals(element))
            b = true;
        else
            compt++;
    }
    return b;
}
```

Cette méthode est basée sur le même principe que la précédente à la différence qu'elle va renvoyer un booléen. Elle fait également appel à la méthode Equals() de la classe de l'élément et elle va renvoyer true si elle trouve l'élément dans le tableau et false sinon.

- Méthode triCroissant et triDecroissant

```
public void TriCroissant()
{
    T temp;
    int compt = nb;
    do
    {
        for (int i = 1; i < nb; i++)
        {
            if (tab[i - 1].CompareTo(tab[i]) >=0)
            {
                temp = tab[i];
                tab[i] = tab[i - 1];
                tab[i - 1] = temp;
            }
        }
        compt--;
    } while (compt != 0);
}
```

Cette méthode trie le tableau par ordre croissant. Pour cela, elle parcourt le tableau n fois (n étant le nombre d'éléments du tableau) et va comparer 2 à 2 ses valeurs. Elle fait appel à la méthode CompareTo() de la classe de l'élément. Il était donc important pour cette méthode d'imposer que l'élément devait avoir implémenté l'interface IComparable<>. Si la valeur 1 est supérieure à la valeur 2 on les échange de place dans le tableau. Pour faire cela il est impératif de définir une variable temporaire. La méthode tri décroissant fonctionne exactement sur le même principe sauf qu'au lieu de vérifier que la valeur renvoyée par CompareTo() est >=0 , cette fois on vérifie que c'est <=0.

- Méthode Affiche

```
public void Affiche()
{
    for (int i = 0; i < nb; i++)
        Console.WriteLine(tab[i].ToString());
}
```

La méthode affiche permet d'afficher le Panier dans la console. Pour cela elle parcourt simplement le tableau d'éléments et les affiche un par un. Elle fait appel à la méthode ToString() de la classe de l'élément.

Je ne vais pas réexpliquer la classe Skin puisqu'elle est quasiment identique à celle de la partie 1 du projet (à la différence que j'ai remis les méthodes que j'avais déplacées dans ma classe parent Equipement dans la classe Skin). Je l'ai simplement recopiée dans le même namespace de cette seconde application.

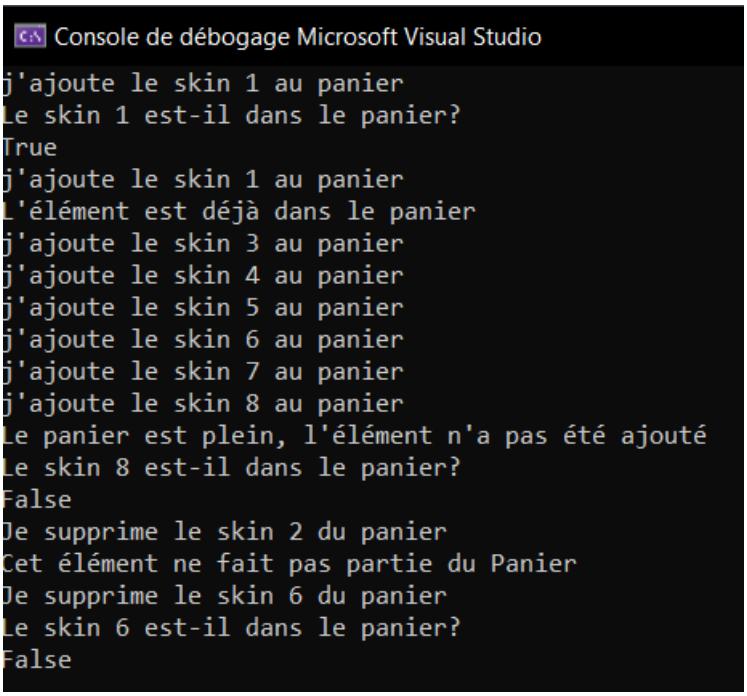
III. Phase de Tests

Dans le programme principal j'ai écrit une série de tests permettant de vérifier que les différentes méthodes de l'application fonctionnaient correctement et d'afficher les différents messages d'erreur définis dans les méthodes.

1. *Test des méthodes d'ajout/suppression/recherche*

```
Panier <Skin> Ps= new Panier<Skin>();
Skin s1 = new Skin();
s1.Nom = "toto1";
Skin s2 = new Skin();
s2.Nom = "toto2";
Skin s3 = new Skin();
s3.Nom = "toto3";
Skin s4 = new Skin();
s4.Nom = "toto4";
Skin s5 = new Skin();
s5.Nom = "toto5";
Skin s6 = new Skin();
s6.Nom = "toto6";
Skin s7 = new Skin();
s7.Nom = "toto7";
Skin s8 = new Skin();
s8.Nom = "toto8";

Console.WriteLine("j'ajoute le skin 1 au panier");
Ps.Ajout(s1);
Console.WriteLine("Le skin 1 est-il dans le panier?");
Console.WriteLine(Ps.Appartient(s1));
Console.WriteLine("j'ajoute le skin 1 au panier");
Ps.Ajout(s1);


Console.WriteLine("j'ajoute le skin 3 au panier");
Ps.Ajout(s3);
Console.WriteLine("j'ajoute le skin 4 au panier");
Ps.Ajout(s4);
Console.WriteLine("j'ajoute le skin 5 au panier");
Ps.Ajout(s5);
Console.WriteLine("j'ajoute le skin 6 au panier");
Ps.Ajout(s6);
Console.WriteLine("j'ajoute le skin 7 au panier");
Ps.Ajout(s7);
Console.WriteLine("j'ajoute le skin 8 au panier");
Ps.Ajout(s8);

Console.WriteLine("Le skin 8 est-il dans le panier?");
Console.WriteLine(Ps.Appartient(s8));

Console.WriteLine("Je supprime le skin 2 du panier");
Ps.Supprime(s2);

Console.WriteLine("Je supprime le skin 6 du panier");
Ps.Supprime(s6);

Console.WriteLine("Le skin 6 est-il dans le panier?");
Console.WriteLine(Ps.Appartient(s6));
```

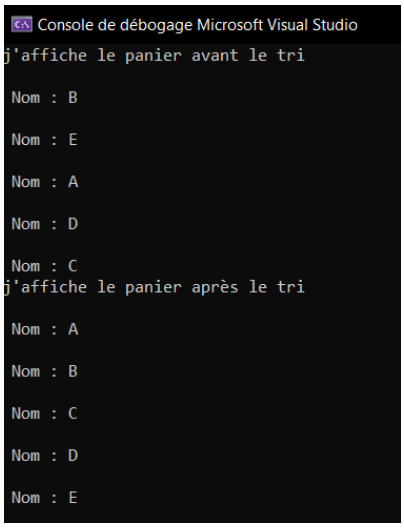
Et voilà ce que nous renvoie la console :



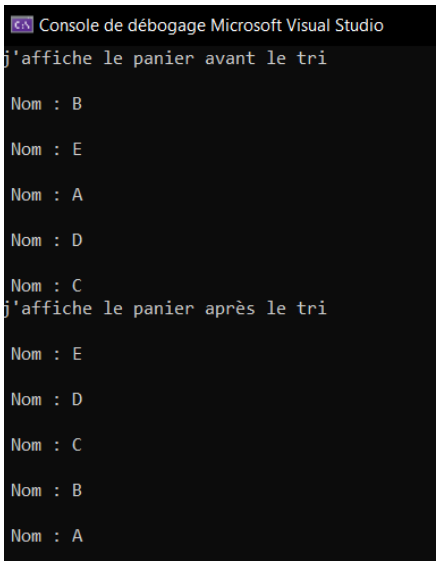
2. Test de la fonction TriCroissant et Affiche

Pour simplifier la compréhension des tests, j'ai modifié la méthode ToString() de la classe Skin pour qu'elle affiche uniquement le nom. Comme le tri se fait en fonction du nom, la compréhension des résultats est plus rapide.

```
Panier<Skin> Ps2 = new Panier<Skin>();
Skin sa = new Skin();
sa.Nom = "A";
Skin sb = new Skin();
sb.Nom = "B";
Skin sc = new Skin();
sc.Nom = "C";
Skin sd = new Skin();
sd.Nom = "D";
Skin se = new Skin();
se.Nom = "E";

Ps2.Ajout(sb);
Ps2.Ajout(se);
Ps2.Ajout(sa);
Ps2.Ajout(sd);
Ps2.Ajout(sc);

Console.WriteLine("j'affiche le panier avant le tri");
Ps2.Affiche();
Console.WriteLine("j'affiche le panier après le tri");
Ps2.TriCroissant();
Ps2.Affiche();
```



Si au lieu d'appeler la méthode triCroissant() on appelle triDecroissant() on obtient :



IV. Annexes

Sites internet sources utilisés : https://docs.microsoft.com