Interfaces

E. Temam

Le problème

- On s'intéresse à la modélisation d’un bricoleur qui peut effectuer certaines tâches telles que visser, couper, casser. Chacune de ces tâches s’accomplit à l’aide d’un outil adapté.
- Par exemple, un tournevis est un outil adapté pour visser, on pourrait donc avoir quelque chose ressemblant à:

```
public class Tournevis
   {
       public void visse()
       {
           Console.WriteLine("Tournevis visse");
       }
   }
```

```
public class Bricoleur
   {
       public void visse(Tournevis t)
       {
           t.visse();
       }
       public void casse(Marteau m)
       {
           m.casse();
       }
       public void coupe(Scie s)
       {
           s.coupe();
       }
   }
   public class Scie
   {
       public void coupe()
       {
           Console.WriteLine("Scie coupe");
       }
   }
   public class Marteau
   {
       public void casse()
       {
           Console.WriteLine("Marteau casse");
       }
   }
```

Prise en compte d’un cutter, d’une masse?

```
public class Cutter
{
    public void coupe()
    {
        Console.WriteLine("Cutter coupe");
    }
}
```

```
public class Masse
{
    public void casse()
    {
        Console.WriteLine("Masse casse");
    }
}
```

```
public class Bricoleur
    {
        public void visse(Tournevis t)
        {
            t.visse();
        }
        public void casse(Marteau m)
        {
            m.casse();
        }
        public void coupe(Scie s)
        {
            s.coupe();
        }
        public void coupe(Cutter c)
        {
            c.coupe();
        }
        public void casse(Masse m)
        {
            m.casse();
        }
    }
```

NON

On doit modifier le code de `Bricoleur` pour ajouter un nouvel outil.

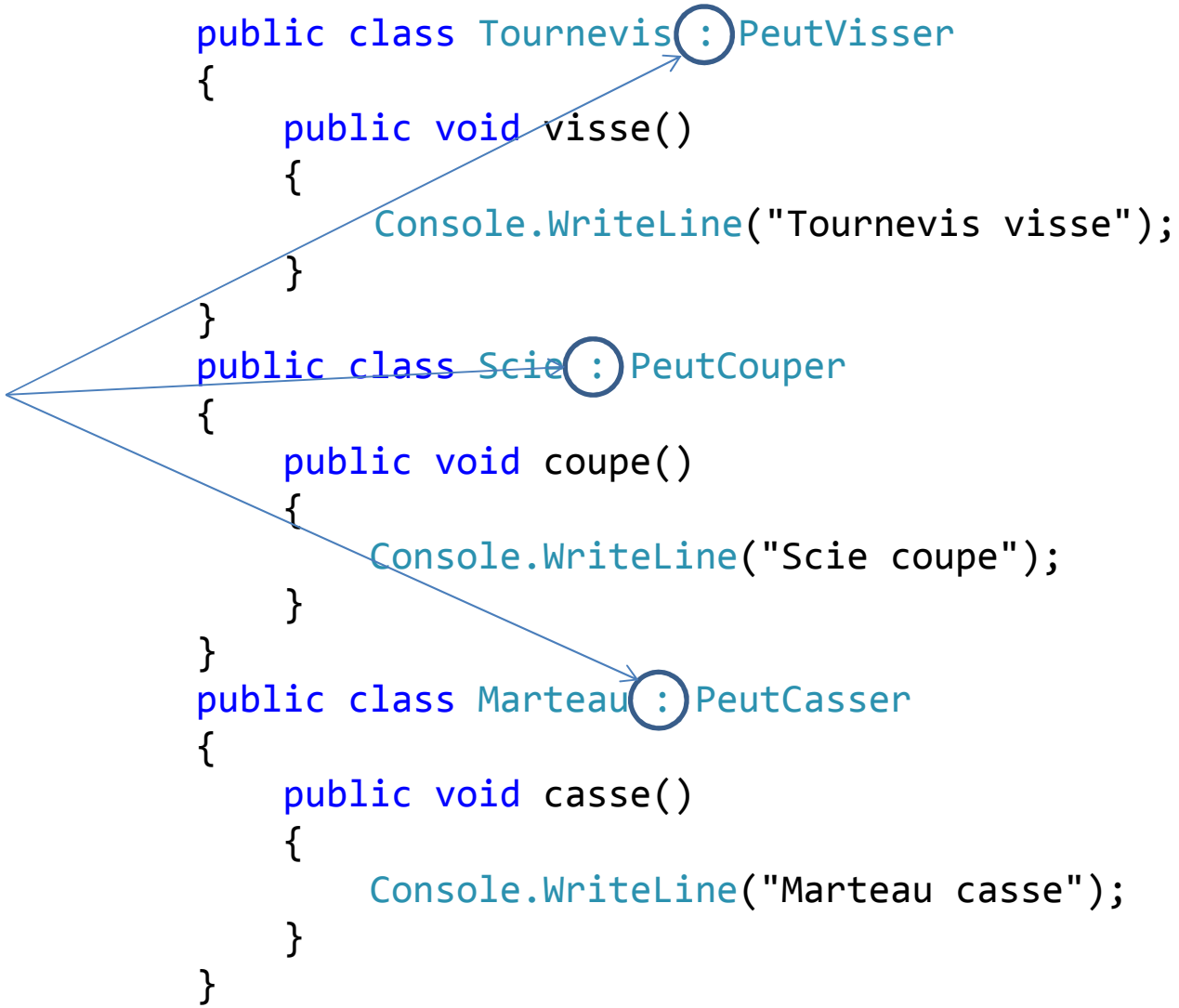
Utiliser les interfaces

- Définir une interface pour les outils sachant couper, visser, casser
- Définir des abstractions pour ces notions

```
public interface PeutVisser
{
    public void visse();
}
public interface PeutCouper
{
    public void coupe();
}
public interface PeutCasser
{
    public void casse();
}
```

Ce qui donne

Comme l'héritage



```
public class Tournevis : PeutVisser
{
    public void visse()
    {
        Console.WriteLine("Tournevis visse");
    }
}
public class Scie : PeutCouper
{
    public void coupe()
    {
        Console.WriteLine("Scie coupe");
    }
}
public class Marteau : PeutCasser
{
    public void casse()
    {
        Console.WriteLine("Marteau casse");
    }
}
```

Et donc,

```
public class Bricoleur
   {
       public void visse(PeutVisser pv)
       {
           pv.visse();
       }
       public void casse(PeutCasser pca)
       {
           pca.casse();
       }
       public void coupe(PeutCouper pco)
       {
           pco.coupe();
       }
   }
```

Et si maintenant, on ajoute

```
public class Masse: PeutCasser
{
    public void casse()
    {
        Console.WriteLine("Masse casse");
    }
}
```

```
public class Cutter : PeutCouper
   {
       public void coupe()
       {
           Console.WriteLine("Cutter coupe");
       }
   }
```

Sans rien modifier on peut écrire:

```
Bricoleur bob = new Bricoleur();
bob.coupe(new Scie());
bob.coupe(new Cutter());
bob.casse(new Marteau());
bob.casse(new Masse());
```

Multi-implémentation

```
public class CouteauSuisse : PeutCasser,PeutCouper,PeutVisser
    {
        public void visse()
        {
            Console.WriteLine("CouteauSuisse visse");
        }
        public void coupe()
        {
            Console.WriteLine("CouteauSuisse coupe");
        }
        public void casse()
        {
            Console.WriteLine("CouteauSuisse casse");
        }
    }
```

```
PeutCouper pc = cs;                  //Upcast de CouteauSuisse vers PeutCouper
pc.coupe();                          //pas de pb
cs.casse();                          //pas de pb
pc.casse();                          //illegal
((CouteauSuisse)pc).casse();         //Downcast licite de PeutCouper vers CouteauSuisse
((Marteau)pc).casse();               //DownCast illicite (compile quand meme) de
                                     //PeutCouper vers Marteau
```

- On veut pouvoir ranger les différents outils dans une boite a outils représentée par un tableau.
- Solution : avoir une interface `Tool` qui sert uniquement a repérer les outils (typer)

Définition

- Une interface en C# est un contrat:
  - Elle peut contenir des propriétés et des méthodes ou des indexeurs mais ne doit contenir aucun attribut
  - Une interface ne peut contenir de méthodes déjà implémentées.
- Une interface ne contient que des signatures
- Tous les membres d'une interface sont public
- Une interface est héritable
- Tous les membres doivent être implémentées dans la classe qui dérive de l'interface

- Interaction avec un objet par envoi de messages (=appel de méthodes)
- Pour manipuler un objet il faut et il suffit de connaitre les messages qu'il accepte
- Type définit l'ensemble des messages acceptés par un objet de ce type.

2 notions:

- Les messages acceptés (intervient à la programmation): signature des méthodes
- La réaction aux messages: (intervient à l'exécution): code des méthodes
- La compilation vérifie la légalité d'un envoi de message sur une référence en fonction de son type

- En C#, pour définir une référence il faut imposer son type
- Mais la référence doit préalablement être initialisée par une instance d'une classe
- Le traitement provoque par l'invocation est alors défini par la classe de l'objet référencé

  une classe est un type, une interface aussi
- Mais en + une classe impose le traitement associé aux messages
- Toutes les instances d'une même classe accomplissent le même traitement

Et?

- Comment permettre des comportements différents pour une même manipulation
  - Avoir des réactions différentes pour une même invocation
  - Séparer l'envoi des messages du traitement associé
  - Séparer la signature de la méthode du code associé

Pourquoi?

- Disposer d'une méthode générique pour le tri d'un tableau. Il faut pouvoir:
  - Typer les éléments du tableau
  - Comparer deux éléments
- Disposer d'un outil de manipulation d'images de différents formats. Pour une image, on veut pouvoir connaitre:
  - Sa taille
  - La valeur du pixel a des coordonnées données
  - Sauvegarder/charger l'image depuis un fichier
  - Les traitements vont dépendre du format mais on veut manipuler les images d'une manière identique et on veut pouvoir ajouter des formats

Tri croissant d'un tableau d'entier

```
public void triCroissant (int[] aTrier)
    {
       int tmp;
       for (int i = aTrier.Length - 1; i > 0; i--)
          for (int j = 0; j < i; j++)
             if (aTrier[j] > aTrier[j + 1])
             {
                tmp = aTrier[j];
                aTrier[j] = aTrier[j + 1];
                aTrier[j + 1] = tmp;
             }
    }
```

Tri décroissant d'un tableau d'entier

```
public void triDecroissant (int[] aTrier)
    {
       int tmp;
       for (int i = aTrier.Length - 1; i > 0; i--)
          for (int j = 0; j < i; j++)
             if (aTrier[j] < aTrier[j + 1])
             {
                tmp = aTrier[j];
                aTrier[j] = aTrier[j + 1];
                aTrier[j + 1] = tmp;
             }
    }
```

Tri alphabétique croissant d'un tableau de chaines.

```
public void triCroissant (string[] aTrier)
    {
        string tmp;
        for (int i = aTrier.Length - 1; i > 0; i--)
            for (int j = 0; j < i; j++)
                if (aTrier[j].CompareTo(aTrier[j + 1])>0)
                {
                    tmp = aTrier[j];
                    aTrier[j] = aTrier[j + 1];
                    aTrier[j + 1] = tmp;
                }
    }
```

Tri croissant selon le poids d'un tableau de carottes

```
public class Carotte
  {
     private int poids;
     public int CompareTo(Carotte c)
     {
        return this.poids - c.poids;
     }
  }
```

```
public void triCroissant(Carotte[] aTrier)
     {
        Carotte tmp;
        for (int i = aTrier.Length - 1; i > 0; i--)
           for (int j = 0; j < i; j++)
              if (aTrier[j].CompareTo(aTrier[j + 1]) > 0)
              {
                 tmp = aTrier[j];
                 aTrier[j] = aTrier[j + 1];
                 aTrier[j + 1] = tmp;
              }
     }
```

Interface générique

```
public interface Comparable<T>
{
    public int CompareTo(T t);
}
public class Carotte : Comparable<Carotte>
{
    private int poids;
    public int CompareTo(Carotte c)
    {
        return this.poids - c.poids;
    }
}
```

```
static public void triBulle<T>(T[] aTrier) where T : Comparable<T>
{
        T tmp;
        for (int i = aTrier.Length - 1; i > 0; i--)
            for (int j = 0; j < i; j++)
                if (aTrier[j].CompareTo(aTrier[j + 1]) > 0)
                {
                    tmp = aTrier[j];
                    aTrier[j] = aTrier[j + 1];
                    aTrier[j + 1] = tmp;
                }
}
static void Main(string[] args)
{
        Carotte[] tabCarotte = { new Carotte(), new Carotte() };
        triBulle<Carotte>(tabCarotte);
}
```

Manipulation d'images

- Application ImageManipulateur de manipulation d'images de differents formats (jpeg, giff, bmp,…)
- Pour une image, on veut pouvoir connaitre:
  - Sa taille
  - La valeur du pixel a des coordonnées données
  - Sauvegarder/charger l'image depuis un fichier
  - Les traitements vont dépendre du format mais on veut manipuler les images d'une manière identique et on veut pouvoir ajouter des formats

NON

```
public class Image
{
   private string type;
   public Image(string Type)
   {
      type = Type;
   }
   public int save()
   {
      if (type.Equals("jpg"))
      {
         return 0;
      }
      else if (type.Equals("bmp"))
      {
         return 0;
      }
      else
         return -1;
}
public class ImageManipulateur
{
   public int saveImage(Image img)
   {
      return img.save();
   }
}
```

Problèmes:

- Ajouter un type: modifier la classe image
- Choix de l'attribut type
- Nécessite de séparer le traitement spécifique aux images

NON

N classes différentes par type d'images

```
public class ImageJPG{}
public class ImageBMP{}

public class ImageManipulateur
{
    public int saveImage(Object img)
    {
        if (img.GetType() == typeof(ImageBMP))
        {
            return 0;
        }else if(img.GetType() == typeof(ImageJPG))
            return 0;
        else
            return -1;
    }
}
```

Problèmes:

- Ajouter un type: modifier la classe ImageManipulateur
- On perd le typage dans saveImage
- Nécessite de séparer le traitement spécifique aux images

Conclusion

- Il faut mixer les approches:
  - Il faut un type commun
  - Il faut des classes différentes pour chaque type d'images
- Solution: INTERFACES
  - Fixent les messages acceptés/autorisés
  - Le comportement doit être implémentés dans chaque classe

OUI

```
public interface Image
  {
     public int save();
  }
  public class ImageJPG : Image
  {
     public int save() { return 0; }
  }
  public class ImageBMP : Image
  {
     public int save() { return 0; }
  }
  public class ImageManipulateur
  {
     public int saveImage(Image img) { return img.save(); }
  }
```

Openclose principle

- Un module doit être ouvert aux extensions mais fermés aux modifications
- Manipuler des abstractions et les concrétiser le plus tard possible
- A l'extrême, commencez par des interfaces et seulement ensuite des classes les implémentant

Exemple: compteur

- Il faut un type Counter
- Les objets doivent accepter:
  - Une initialisation
  - Un increment
  - Renvoyer la valeur courante

```
public interface counter
  {
     public int CurrentValue { get; }
     public void increment();
     public void initValue(int init);
  }
```

Exemples

```
public class SimpleCounter : counter
  {
     private int val;
     public SimpleCounter(int init)
     {
        initValue(init);
     }
     public int CurrentValue
     {
        get { return val; }
     }
     public void increment()
     {
        val = val + 1;
     }
     public void initValue(int init)
     {
        val = init;
     }
  }
```

```
public class ModularCounter : counter
 {
     private int val;
     private int modulo;
     public ModularCounter(int init, int mod)
     {
        modulo = mod;
        initValue(init);
     }
     public int CurrentValue
     {
        get { return val; }
     }
     public void increment()
     {
        val = (val + 1) % modulo;
     }
     public void initValue(int init)
     {
        val = init;
     }
 }
```

L'ajout d'une nouvelle classe se fait simplement…

Abstraction de la notion d'incrément

```
public interface IncrementFunction
  {
     public int increment(int value);
  }
  public class SimpleIncrement : IncrementFunction
  {
     public int increment(int value)
     {
        return value + 1;
     }
  }
  public class ModularIncrement : IncrementFunction
  {
     private int modulo;
     public ModularIncrement(int mod) { modulo = mod; }
     public int increment(int value)
     {
        return (value + 1) % modulo;
     }
  }
  public class AnotherIncrement : IncrementFunction
  {
     public int increment(int value)
     {
        return (2 * value + 1);
     }
  }
```

```
public class Counter
  {
     private int val;
     private IncrementFunction incrementF;
     public Counter(int Value, IncrementFunction F)
     {
        this.val = Value;
        this.incrementF = F;
     }
     public int CurrentValue
     {
        get { return val; }
     }
     public void initValue(int init)
     {
        val = init;
     }
     public void increment()
     {
        val = incrementF.increment(val);
     }
  }

  class Program
  {
     static void Main(string[] args)
     {
        Counter simpleCounter = new Counter(0, new SimpleIncrement());
        Counter modularCounter = new Counter(0, new ModularIncrement(7));
     }
  }
```

Résumé

- Les interfaces sont des types:
  - Fixent les signatures sans imposer le comportement
  - Permettent une vision polymorphe des objets
  - Permettent d'offrir aux autres un cadre de programmation
  - Permettent de réutiliser des classes et de les adapter a un contexte
  - Facilitent l'extension d'un programme