C# Synthèse

Les points clefs

Classe abstraite

- Une classe est dite abstraite lorsque le préfixe abstract est rajouté devant le mot clé class :
  - Une classe abstraite permet de déclarer des membres abstraits
    - Methodes ou champs précédés aussi par abstract
      - => sans corps de fonction

```
abstract class A_Fiche
{
protected int valeur=10;

public void test()
{ Console.WriteLine("test"); }

abstract public int AccessValeur
 { get; set; } // PAS DE CORPS!!!!

}
```

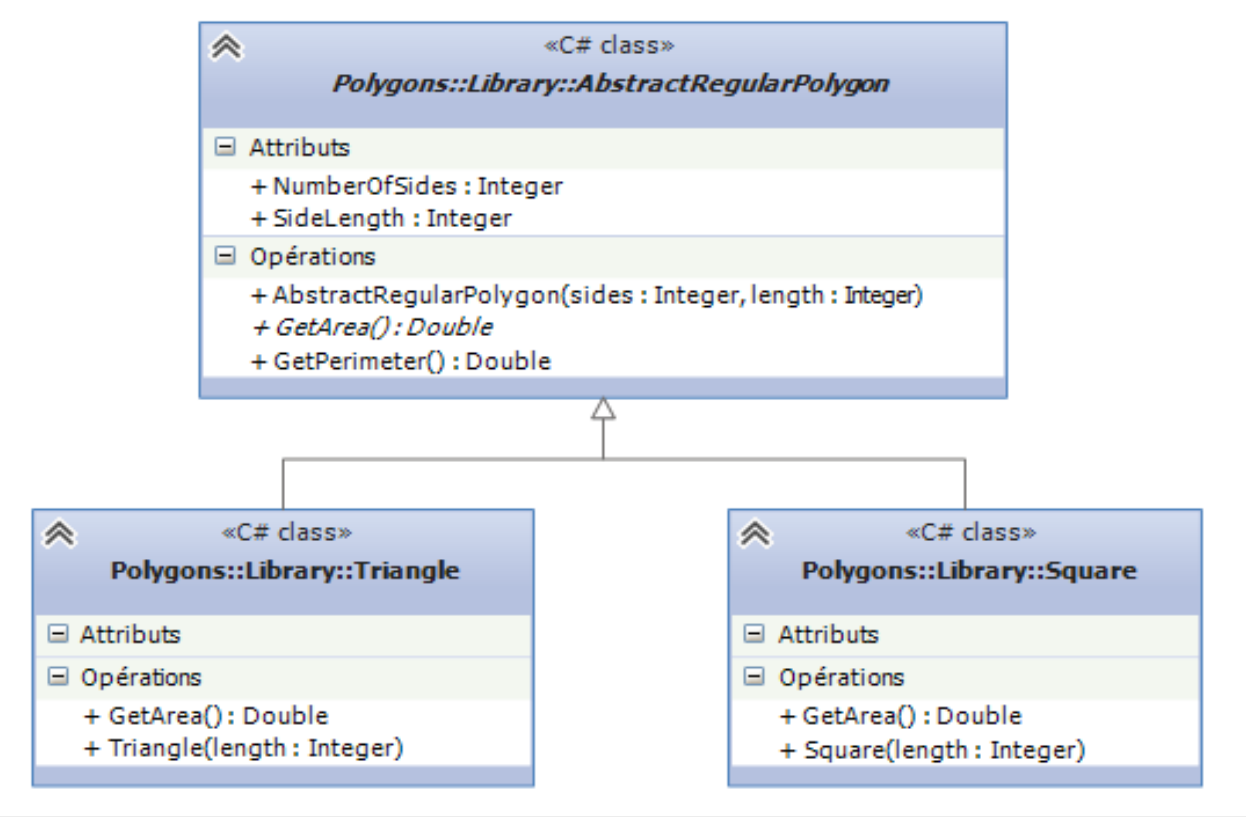
```
class Fiche1 : A_Fiche1
 {
//OBLIGATION d'IMPLENTER l'ACCESSEUR!
 public override int AccessValeur
 {
 get { return (valeur); }
 set { valeur = value; }
 }
```

Classe abstraite

- Une classe abstraite n'est pas instantiable!
  - peut être utilisée uniquement si une classe nouvelle est dérivée d'elle
- Les classes abstraites sont un outils puissants de modélisation
  - Définition en amont du projet des fonctionnalités et interactions des différents objets
  - Elles permettent un découplage entre la définition et l'implémentation d'un objet
  - Définisse un cadre commun pour les équipes de développement
- Le polymorphisme s’applique sur la classe de base abstraite

Classe abstraite

- Pierre angulaire OO
  - Polymorphisme + Classe abstraite



Classe abstraite

- Code client

```
static void Main(string[] args)
        {
            List<AbstractRegularPolygon> listeShape = new List<AbstractRegularPolygon>();
            listeShape.Add(new Square(1));;
            listeShape.Add(new Triangle(1));
            //liste.Add(new AbstractRegularPolygon(5));  KO pas
possible
            foreach (AbstractRegularPolygon shape in listeShape)
            {
                DisplayPolygon(shape);
            }
            Console.ReadKey();
        }

        public static void DisplayPolygon(AbstractRegularPolygon polygon)
        {

            Console.WriteLine("Number of Sides:{0}", polygon.NumberOfSides);
            Console.WriteLine("Side Length:{0}", polygon.SideLength);
            Console.WriteLine("Perimeter:{0}", polygon.GetPerimeter());
            Console.WriteLine("Area:{0}", Math.Round(polygon.GetArea(), 2));
            Console.WriteLine();
        }
```

Les polygones sont stockés et manipulés au travers de la classe de base : AbstractRegularPolygon

Manipulation par le biais d'une référence de type AbstractRegularPolygon

Polymorphisme en action : la méthode correspond au vrai type est retrouvé et effectuée

Interfaces

- Concepts
  - Une **interface** peut être vue donc comme un modèle de classe qui contient *les prototypes* des méthodes et des propriétés
    - aucune définition de corps: tous les éléments sont abstraits
  - *Chaque classe qui choisira d'implémenter l'interface devra respecter le contrat imposé et fournir une implémentation de l'interface* (ie fournir le code des méthodes
  - **Une classe peut implémenter plusieurs interfaces**
  - Les interfaces sont polymorphes

Interfaces

- Quand utiliser des interfaces

Classes abstraites	Interfaces
Peut contenir du code	Ne contient aucun code
Un seul héritage à la fois possible (si j'hérite dela classe abstraite il n'est plus possible d'hériter d'une autre classe)	Une classe peut implémenter plusieurs interfaces
Les membres de la classes ont des modificateurs d'accès (public, private..)	Tous les membres de la classes sont public
Les membres peuvent être des: ▪ propriétés ▪ méthodes ▪ evénement (event) ▪ indexeurs ▪ constructeurs/destructeurs ▪ champs	Les membres peuvent être des: ▪ propriétés ▪ méthodes ▪ evénement (event) ▪ indexeurs

Les interfaces sont à reserver:

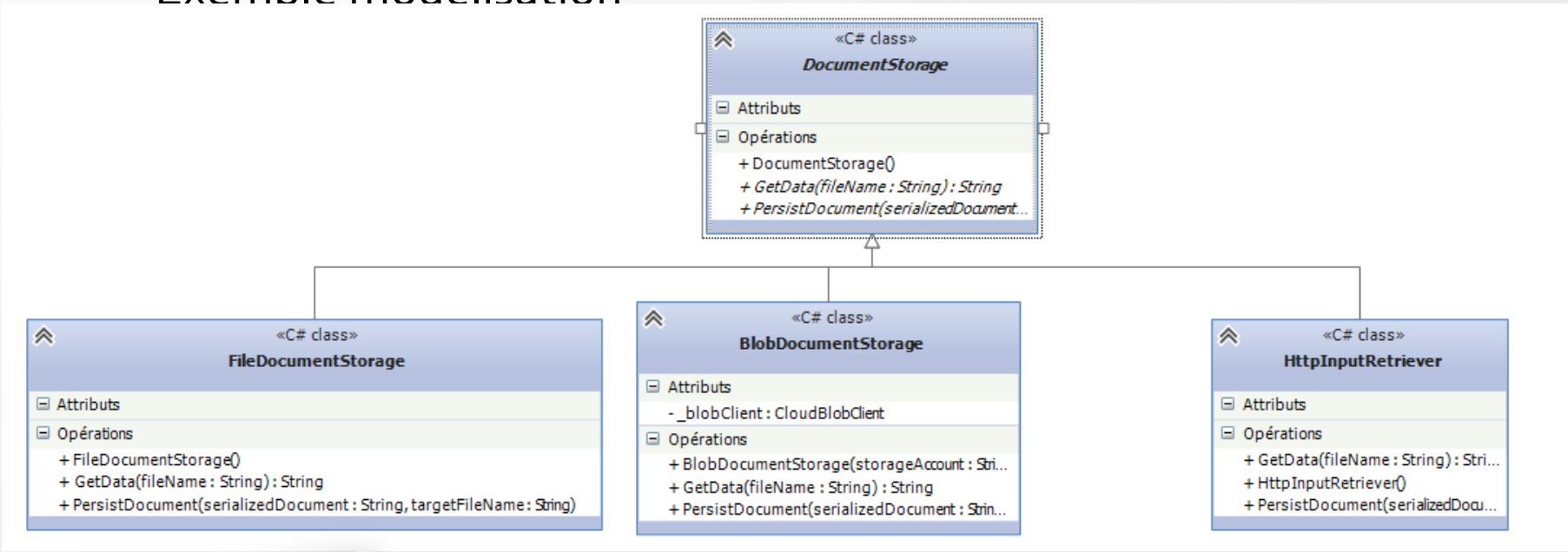
pour des fonctionnalités connexes qui peuvent appartenir à n'importe quelle classe

Si la fonctionnalité que vous créez est utile à de nombreux objets différents

Les interfaces sont plutôt petites et indépendantes les unes des autres.

Interfaces

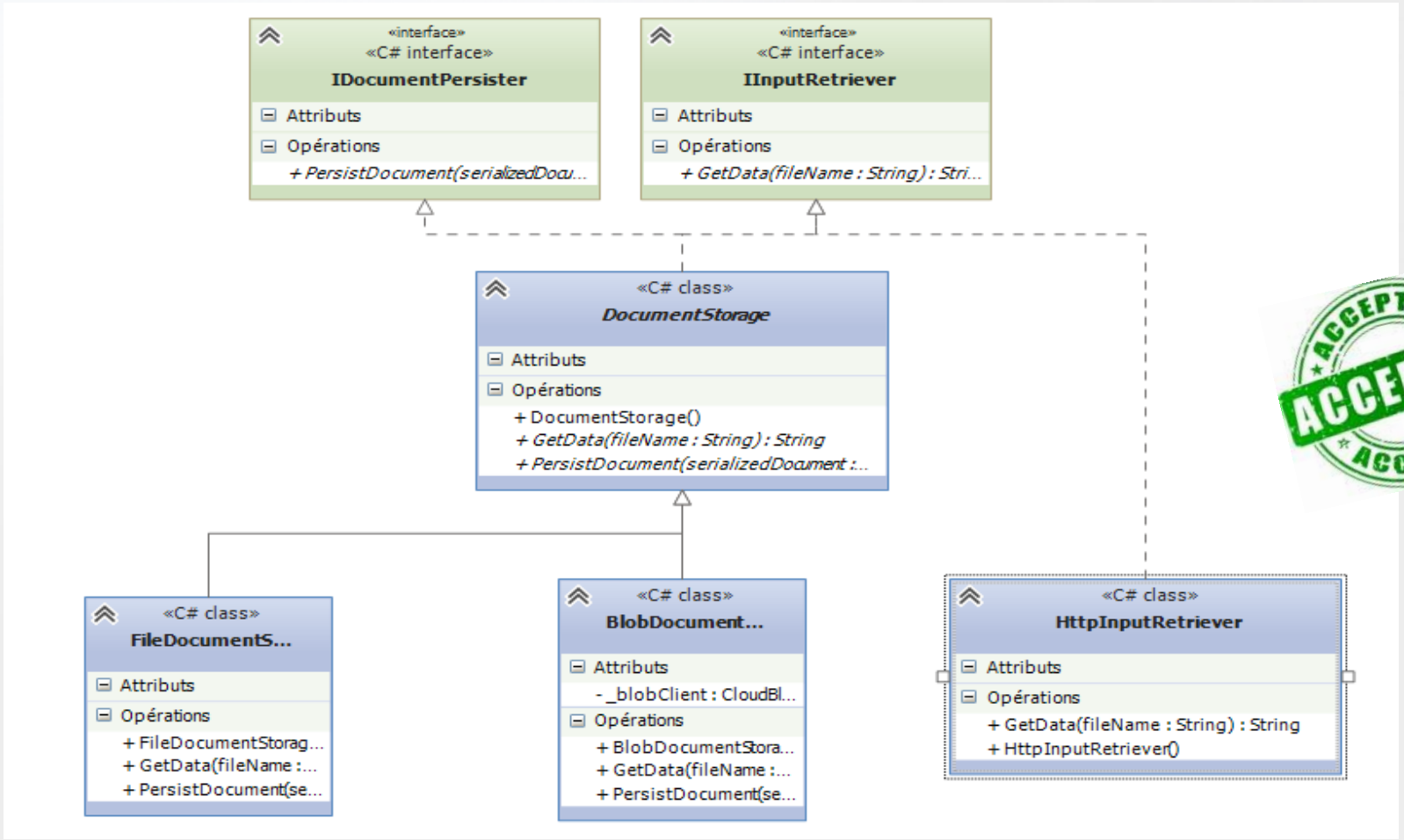
- Exemple modélisation




```
public class HttpInputRetriever : DocumentStorage
{
    4 références
    public override void PersistDocument(string serializedDocument, string targetFileName)
    {
        // Impossible to solve...
        // Fixed in the ISP phase...
        throw new NotImplementedException();
    }
```

Problème de comportement
=> Résolution en changeant la modélisation

Interfaces


«interface»
«C# interface»
IDocumentPersister
Attributs
Opérations
+ PersistDocument(serializedDocu...
«interface»
«C# interface»
IInputRetriever
Attributs
Opérations
+ GetData(fileName : String) : Stri...
«C# class»
DocumentStorage
Attributs
Opérations
+ DocumentStorage()
+ GetData(fileName : String) : String
+ PersistDocument(serializedDocument :...
«C# class»
FileDocumentS...
Attributs
Opérations
+ FileDocumentStorag...
+ GetData(fileName :...
+ PersistDocument(se...
«C# class»
BlobDocument...
Attributs
- _blobClient : CloudBl...
Opérations
+ BlobDocumentStora...
+ GetData(fileName :...
+ PersistDocument(se...
«C# class»
HttpInputRetriever
Attributs
Opérations
+ GetData(fileName : String) : String
+ HttpInputRetriever()
ACCEPTED
ACCEPTED
ACCEPTED

Interfaces

- Codage
  - Mise ne place des définitions

```
//Fonction Vole
  interface IVolant
  {void Vole();}
  // Fonction a des pattes
  interface IPatte
  {   // définition d'une propriété en lecture
   int NbPatte { get; }
  }
  // classe de base
  public class Mammifere
  {
  //methodes communes à tous les mammifères
  public virtual void Crier() { C
           onsole.WriteLine("Raaaa"); }
  public virtual void Sous_Ordre_Especes() {
           Console.WriteLine("Mammifere"); }
 }
```

```
// Chien est un mammifère qui a aussi des pattes
    class Chien : Mammifere,IPatte
    {  // Implantation de NbPatte défini dans IPatte
        public int NbPatte
             {   get {return(2);}
             }
        // surcharge polymorphe des classes de bases
        public override void Crier() { Console.WriteLine("Ouarfff"); }
        public override void Sous_Ordre_Especes() {
             Console.WriteLine("Canidés"); }
     }
class Baleine : Mammifere
{   // surcharge polymorphe des classes de bases
    public override void Crier() { Console.WriteLine("Mouuuuu"); }
    public override void Sous_Ordre_Especes() {
Console.WriteLine("Cétacés");}
}
// Chien est un mammifère avec les fonctionnalités: pattes + volant
class ChauveSouris : Mammifere,IVolant,IPatte
{   // Implantation de NbPatte défini dans IPatte
    public int NbPatte
         {   get {return(2);}
         }
    // Implantation de Vole défini dans IVolant
    public void Vole() { Console.WriteLine("Flap flap"); }
    // surcharge polymorphe des classes de bases
    public override void Crier() { Console.WriteLine("Mriiiiii"); }
    public override void Sous_Ordre_Especes() {
             Console.WriteLine("chiroptères");}

}
```

Interfaces

- Un exemple
  - Utilisation des classes crées

```
ChauveSouris uneChauveSouris = new ChauveSouris();
uneChauveSouris.Vole();
// On regroupe tout le monde sous le type de base
List<Mammifere> maMenagerie = new List<Mammifere>();
// Je peuple la ménagerie
maMenagerie.Add(new ChauveSouris());
maMenagerie.Add(new Baleine());
maMenagerie.Add(new ChauveSouris());
maMenagerie.Add(new Chien());
//Qui vole dans ma ménagerie????
foreach (Mammifere mam in maMenagerie)
{
    if (mam is IVolant)
        ((IVolant)mam).Vole();
    else
        Console.WriteLine("Non Volant");
}
//Allez tous le monde crie
foreach (Mammifere mam in maMenagerie)
    mam.Crier(); // triatement de base crier =>polymorphisme
```

La conversion de type dans vos programme est un indicateur de « design smell »
Solution ici:
Définir une nouvelle liste basée sur une requête d'extraction de tous les volants.

Interfaces

- Interfaces utiles du framework .NET
  - Objectif : formaliser certaines fonctionnalités récurrentes en algorithme
    - Cloner un objet: *ICloneable<T>*
    - *Parcourir un liste d'objet: IEnumerable<T>*
    - *Comparer des objets*
      - *IEquatable<T>, IEqualityComparer<T>*

```
public class Temperature : IEquatable<Temperature>
{
   // CHAMPS
    protected double m_Temperature_Moyenne = 0.0;
    //ACCESSEURS
    public double Kelvin
    {
        get
        { return m_Temperature_Moyenne; }
        set
        {
            if (value < 0.0)
                throw new ArgumentException("Temperature >0 svp");
            else
                m_Temperature_Moyenne = value;
        }
    }
    // CTOR
    public Temperature(double degreesKelvin1, double degreesKelvin2)
    {
        this.Kelvin = (degreesKelvin1 + degreesKelvin2) / 2;
    }
    // Implémentation de Iequatable:<T> de type Temperature
    public bool Equals(Temperature other)
    {   // appel de méthode par défaut Equals des types par valeurs double.Equals
        return m_Temperature_Moyenne.Equals(other.m_Temperature_Moyenne);
    }}
```

```
List<Temperature> temps =
    new List<Temperature>();
temps.Add(new Temperature(2017.15,1000));
temps.Add(new Temperature(0,100));
temps.Add(new Temperature(50,50));
foreach (Temperature t in temps)
  if (t.Equals(temps[1]))
    Console.WriteLine("température moyenne égale");
  else
      Console.WriteLine("température moyenne différente");
```