version 1.0

Formations à l'informatique

Découvrez la différence ENI

Développement en couches

Support de cours

Ressources Informatiques: C# 5 – Les fondamentaux du langage

www.eni-ecole.fr

Découpage du cours

- **Module 1 :**
  - Le modèle en couches des applications d'entreprise
- **Module 2 :**
  - Développement de la couche BO : rappels POO avec C#
- **Module 3 :**
  - Développement de la couche BLL : compléments POO avec C#
- **Module 4 :**
  - Développement de la couche IHM : compléments POO avec C#
- **Module 5**
  - Développement de la couche IHM : composants Windows Forms

Développement en couches

Module 1

Le modèle en couches des applications d'entreprise

Présentation des couches

- Une application d'entreprise est découpée classiquement en quatre niveaux d'abstraction distincts :
  - La **couche présentation** (IHM, Interface Homme Machine)
  - La **couche logique applicative** (BLL, Business Logic Layer)
  - La **couche métier** (BO, Business Objects)
  - La **couche d'accès aux données** (DAL, Data Access Logic)

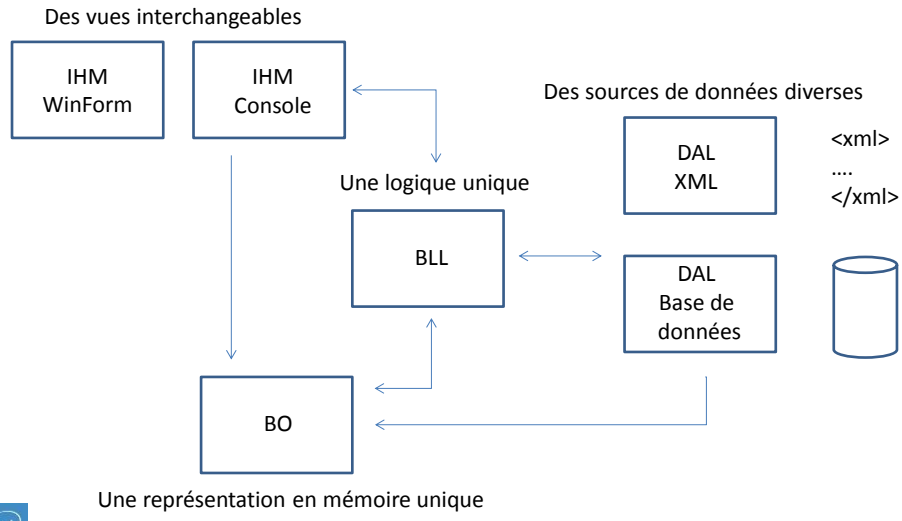
Présentation des couches

- La couche présentation (IHM) :
  - Elle permet à l'utilisateur de dialoguer avec l'application.
  - Elle utilise les entités (BO) et la logique métier (BLL) afin de représenter graphiquement le résultat à l'utilisateur.
- La couche logique applicative (BLL) :
  - Elle décrit ce que fait l'application et comment elle le fait en s'appuyant sur les entités (BO) et les modes d'accès aux données à sa disposition (DAL).
  - Reçoit et analyse les demandes de l'utilisateur.
  - Retrouve et modifie les données via la couche données.
  - Renvoie les résultats à la couche présentation.

Présentation des couches

- La couche métier (BO) :
  - Elle décrit les entités qui structurent le domaine de l'application.
  - Elle structure les données montées en mémoire.
  - Elle assure la sécurité et l'intégrité des données en mémoire.
- La couche d'accès aux données (DAL) :
  - Elle s'occupe de prendre les entités (BO) et de les persister dans une source de données et réalise l'opération inverse.

Présentation des couches



Pourquoi une architecture en couches ?

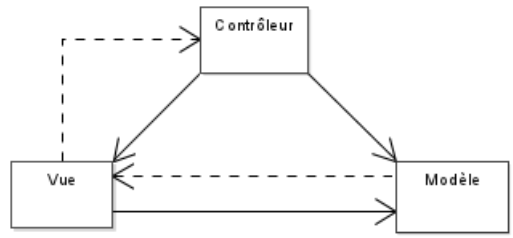
- Pour avoir un modèle de développement commun à toute l'équipe
  - Favorise la modularité, amène à la conception de bibliothèques réutilisables.
  - Le travail en équipe est optimisé : des membres de l'équipe travaillent sur la couche d'accès aux données, un autre peut tout à fait travailler sur la couche métier ou sur l'interface graphique sans perturber le travail de ses collègues.
- Pour faciliter le transfert de compétence
  - La structure des différents projets est standardisée.
  - Le modèle de conception est connu.

Pourquoi une architecture en couches ?

- Pour faciliter la maintenance évolutive et corrective
  - La maintenance des traitements est simplifiée (à chaque couche sa responsabilité).
  - La migration (de l'interface utilisateur par exemple) d'un environnement à un autre est relativement simple : inutile de redévelopper tout ce qui a été fait jusqu'à maintenant, il suffit de modifier l'interface.
- Pour ne pas réinventer la roue à chaque nouveau projet
  - Le modèle d'architecture MVC est la base de nombreux *Frameworks* de développement d'applications quelque soit le langage…
  - Utilisé également dans la conception des composants d'interface utilisateur en mode graphique (Winform, WPF, Swing…)

Architecture inspirée du modèle de conception MVC

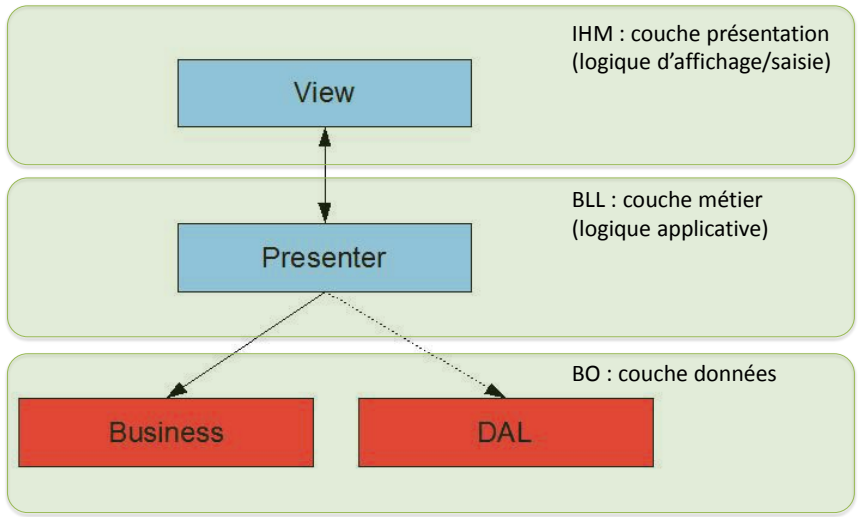
- Modèle de conception classique qui a fait ses preuves.
  - MVC = Modèle – Vue – Contrôleur



  - **Vue** : son but est d'afficher les données du modèle, communique directement avec le modèle.
  - **Contrôleur** : gère les entrées utilisateurs et commande au modèle, communique directement avec le modèle.
  - **Modèle** : ensemble d'objets métiers qui représentent le domaine de l'application, qui implémentent la logique métier. Ils n'ont pas connaissance de l'interface utilisateur (vue, contrôleur).
- Il existe un certain nombre de variantes :
  - MVC, Application Model – MVC, MVP (Modèle – Vue – Présentateur), MVPC (Modèle – Vue –Présentateur - Contrôleur)…

Architecture retenue : principe de base

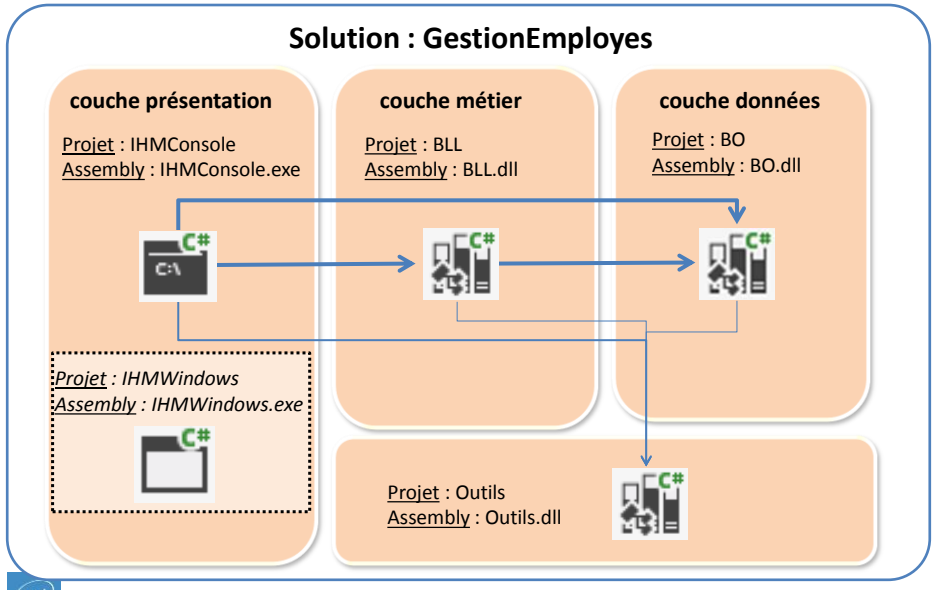
- Elle se base sur une variante de MVC : Modèle - Vue – Présentateur (MVP)



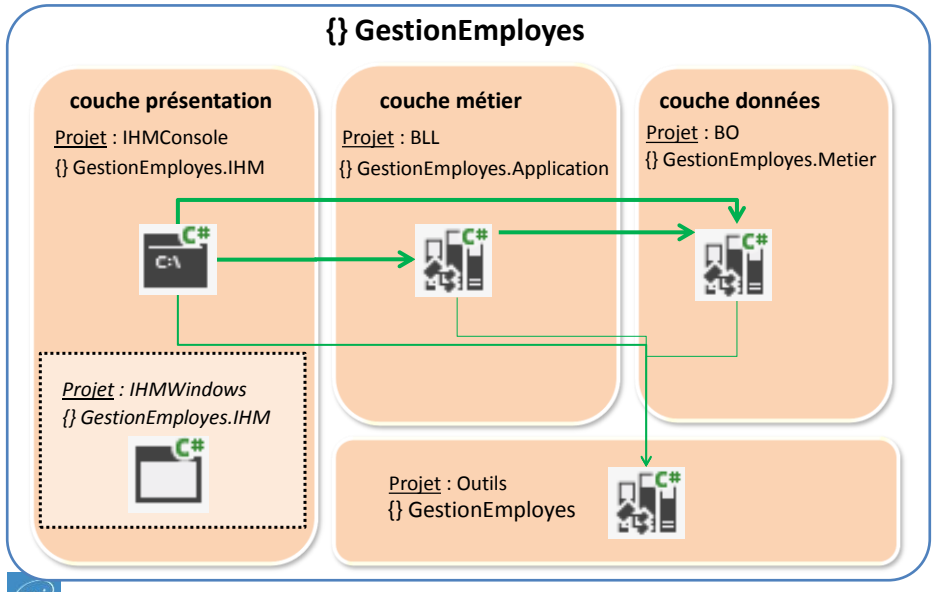
Architecture retenue : spécificités de notre solution

- Adapter le modèle d'application à son contexte d'utilisation :
  - Implémentation en fonction des contraintes et buts de l'application.
- Dans le modèle MVP, le Présentateur pilote la Vue (c'est-à-dire qu'il choisit quelle vue utiliser, comment s'enchaînent les vues, etc.). Ce ne sera pas le cas dans notre solution.
  - Le point d'entrée de l'application sera la couche IHM.
  - C'est au niveau de la couche IHM elle-même que se piloteront les vues.
- Chaque IHM nécessitant une interaction avec des informations issues de la couche de données utilisera un ou plusieurs objets de la couche métier à cet effet.
- Les composants de la couche métier seront « orientés » vers les fonctions de l'application (gestion des services, gestion des employés…)

Architecture retenue : organisation physique

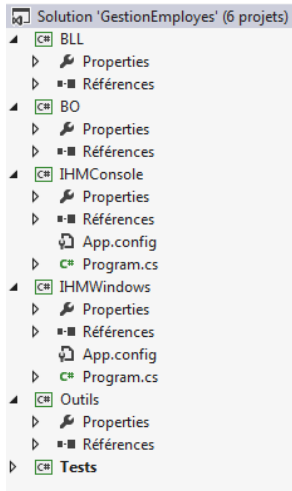


Architecture retenue : organisation logique

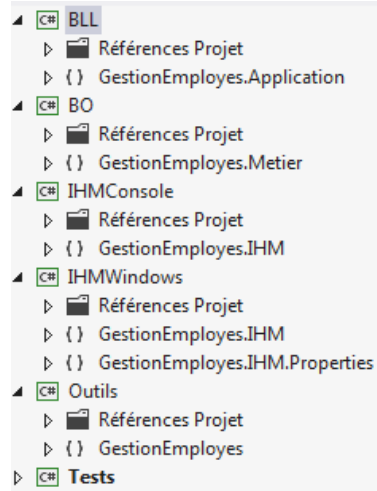


Architecture retenue : implémentation dans VS

- Organisation physique vue par l'*Explorateur de solution* :



- Organisation logique vue par l'*Affichage des classes* :



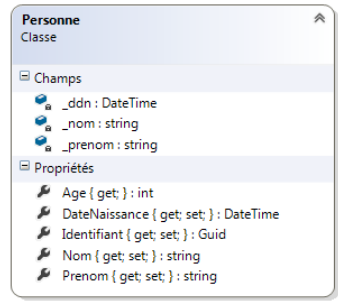
Développement en couches

Module 2

Développement de la couche BO
Rappel POO avec C#

Classe – Création

- **RI 160**
- Diagramme de classe :



- Code correspondant :

```
public class Personne
{

    "Les attributs"

    "Les propriétés"

}
```


Classe - Utilisation

- Utilisation dans le projet **Tests**

```
namespace Tests
{
    class Program
    {
        static void Main(string[] args)
        {
            testPersonne();
        }

        private static void testPersonne()
        {
            Personne p = null;
        }
    }
}
```

Le projet ne connaît pas la classe Personne

00 %

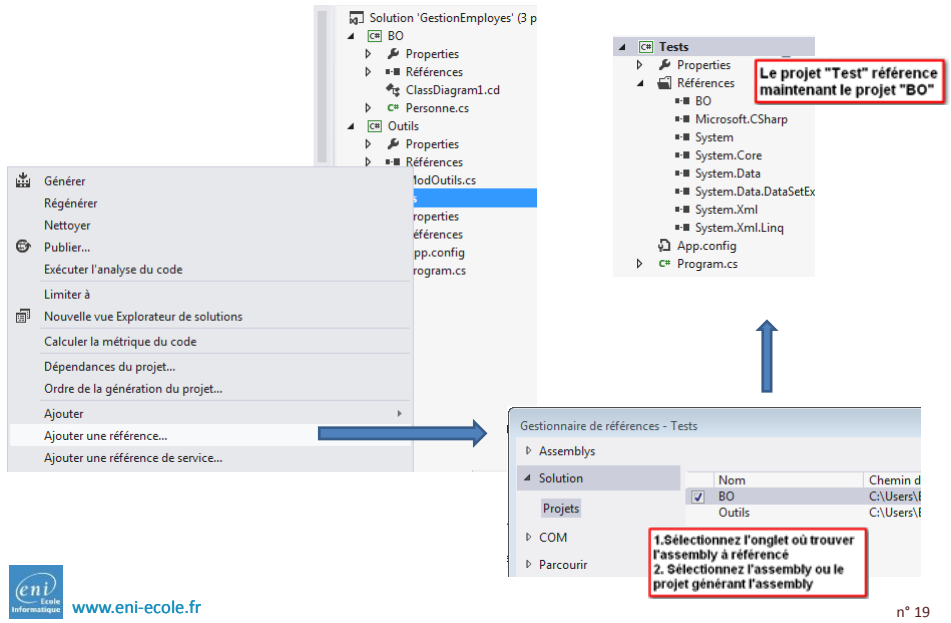
iste d'erreurs

1 erreur | 0 avertissements | 0 messages

Description

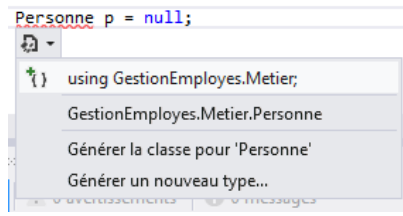
1 Le type ou le nom d'espace de noms 'Personne' est introuvable (une directive using ou une référence d'assembly est-elle manquante ?)

Classe – Définir les références



Classe - Utilisation

- L'auto-complétion nous propose alors d'importer le *namespace* contenant la classe Personne **RI 177**




```
using GestionEmployes.Metier;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Tests
{
    class Program
    {
```

Classe - Utilisation

- Cycle de vie d'un objet en mémoire

```
private static void testPersonne()
{
    /*
     * Variable objet
     */
    Personne p = null;
    /*
     * Instanciation d'un objet en mémoire
     * Utilisation du constructeur par défaut
     * L'objet est référencé par la variable objet
     */
    p = new Personne();
    /*
     * L'objet peut être utilisé
     */
    p.Nom = "HADDOCK";
    p.Prenom = "Archibal";
    p.DateNaissance = new DateTime(1924, 2, 25);
    Console.WriteLine(p.Nom);
    /*
     * Héritage implicite avec la classe Object
     */
    Console.WriteLine(p.GetHashCode() + " - " + p.GetType().Name + " - " + p.ToString());
    /*
     * Déréférencer l'objet : l'objet peut être détruit par le Garbage Collector
     */
    p = null;
```


Propriétés - problématique

- Les attributs publics
  - Accès en lecture/écriture
  - Aucun contrôle des valeurs
  - Aucun formatage

- => La solution est l'encapsulation
  - Mise en œuvre :
    - Rendre les attributs privés
    - Accéder aux attributs via des méthodes
  - Solution .NET :
    - Utilisation des propriétés **RI 162**

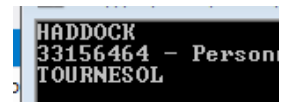
Propriétés – Solution

- Mise en œuvre sur la classe Personne

```
#region "Les attributs"
/**
 * Les attributs
 */
private String _nom;
private String _prenom;
private DateTime _ddn;

#endregion

#region "Les propriétés"
/**
 * Les propriétés d'accès aux attributs
 */
public String Nom
{
    get { return _nom; }
    set { _nom = value.ToUpper(); }
}

public String Prenom...

public DateTime DateNaissance...
```

- Utilisation

```
p = new Personne();
p.Nom = "tournesol";
Console.WriteLine(p.Nom);
```



Propriétés – Lecture seule/écriture seule

```
/**
 * propriété en lecture seule
 */
public int Age
{
    get { return ModOutils.AnneeDiff(this.DateNaissance, DateTime.Now); }
}
```

Propriétés – Propriété auto-implémentée

```
/**
 * propriété auto-implémentée
 */
public Guid Identifiant { get; set; }
```


Constructeur – constructeur par défaut

- **RI 174**
- Pour avoir un objet (ou instance d'une classe), il faut construire l'objet.
  - Existence d'un constructeur par défaut sans paramètre :

```
p = new Personne();
```

  - Ce constructeur existe implicitement si et seulement si aucun constructeur n'est défini dans la classe d'appartenance de l'objet.
- Une fois l'objet instancié, on peut le manipuler.

Constructeur – constructeur défini dans la classe

- On peut personnaliser le constructeur par défaut

```
/// <summary>
/// constructeur par défaut de la classe Personne
/// </summary>
public Personne()
    : base() //appel au constructeur de la super classe (sous entendu)
{
    /*
     * code supplémentaire à executer lors de la construction d'un objet Personne
     */
    this.Identifiant = Guid.NewGuid();
}
```

- Utilisation du mot clé **this** pour accéder aux propriétés.
- Utilisation du mot clé **base** pour faire référence à la classe mère.
- On peut définir plusieurs constructeurs dans une classe

Constructeur – constructeur défini dans la classe

- Lorsqu'un constructeur est défini, le constructeur par défaut n'existe plus. Il faut donc explicitement l'écrire pour pouvoir l'utiliser à nouveau.

```
#region "Constructeurs et destructeur"
/// <summary>
/// constructeur par défaut de la classe Personne
/// </summary>
public Personne()
    : base() //appel au constructeur de la super classe (sous entendu)
{
    /*
     * code supplémentaire à executer lors de la construction d'un objet Personne
     */
    this.Identifiant = Guid.NewGuid();
}

/// <summary>
/// constructeur surchargé
/// </summary>
/// <param name="nom"></param>
/// <param name="prenom"></param>
/// <param name="dateNaissance"></param>
public Personne(String nom, String prenom, DateTime dateNaissance)
    :this() //appel explicite au constructeur par défaut de la classe
{
    /*
     * code supplémentaire à executer lors de la construction d'un objet Personne
     */
    this.Nom = nom;
    this.Prenom = prenom;
    this.DateNaissance = dateNaissance;
}
```

Ajout explicite du constructeur sans paramètre

Ajout d'un constructeur avec paramètres

Constructeur – constructeur défini dans la classe

- Utilisation

```
p = new Personne("tournesol", "Tryphon", new DateTime(1920, 6, 20));
Console.WriteLine("{0} {1} {2}",p.Identifiant, p.Nom,p.Prenom);
```

```
e092ca75-8783-431d-8d07-47bb314c0f76 TOURNESOL Tryphon
```


Destructeur

- La destruction d'un objet en mémoire est prise en charge par le Garbage Collector.
- Il est possible d'écrire explicitement dans une classe, une méthode « branchée » sur ce processus de destruction.
- Cette méthode se nomme le destructeur.

```
/// <summary>
/// Destructeur de la classe Personne
/// Appelé implicitement par le Garbage Collector
/// </summary>
~Personne()
{
    /*
     * code exécuté implicitement lors de la destruction
     * de l'objet en mémoire par le Garbage collector
     */
}
```

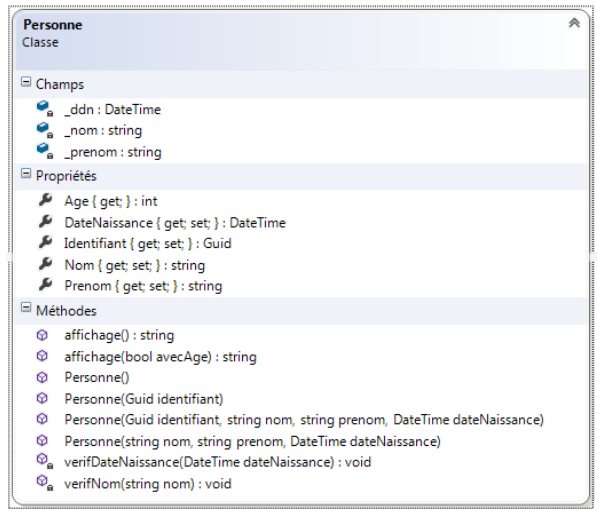
Destructeur et méthode Dispose()

- Une convention d'écriture.
- Permet d'appeler explicitement le code de libération des ressources.

```
public void dispose()
{
    /*
    * code à exécuter explicitement lors du déréférencement d'un objet
    */

    /*
     * Signaler au Garbage Collector que le travail à déjà été fait
     * pour cette instance
     */
    GC.SuppressFinalize(this);
}
```


Méthodes

- **RI 168**
- La classe Personne expose ces comportements au travers de méthodes.
- La classe Personne va être enrichie de méthodes :



Méthodes privées

- Pré-requis:
  - Comment lever une exception? **RI 228**

```
Throw New ApplicationException("Le message de mon exception")
```

- Consigne:
  - Ecrire la méthode verifNom()
    - Comportement : lève une exception si la vérification n'est pas bonne.
    - Contrainte : le nom est obligatoire

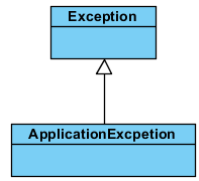
```
/// <summary>
/// Vérifie la validité d'une chaine proposée pour le nom d'uine instance de Personne
/// </summary>
/// <param name="nom">chaine à vérifier</param>
private void verifNom(String nom)
{
    if (String.IsNullOrEmpty(nom))
        throw new ApplicationException("Le nom ne peut pas être vide ou null");
}
```


Méthodes de vérification- Branchement

```
public String Nom
{
    get { return _nom; }
    set {
        /*
         * Valider les données avant de modifier l'état
         */
        this.verifNom(value);
        _nom = value.ToUpper();
    }
}
```

Gestion des exceptions

- Le bloc try/catch **RI 228**
  - Une exception est typée.
  - Le code susceptible de lever une erreur doit être placé dans un bloc **try**.
  - Les blocs **catch** s'occupent de traiter les erreurs selon leur type.
  - Les blocs **catch** doivent être placés dans un ordre précis : du type le plus spécifique au type le plus général.
  - Un seul des blocs **catch** est utilisé pour traiter une exception :
    - Exemple :
      - VerifNom() peut lever une ApplicationException.
      - **ApplicationException** est un sous-type de **Exception**



      - Le premier bloc ApplicationException ou Exception rencontré traitera l'erreur

Exception – Mise en œuvre

- Exemple de gestion d'une exception

```
try
{
    p.Nom = "";
}
catch (ApplicationException ae)
{
    Console.WriteLine(ae.Message);
}
/*
 * l'état de l'objet p n'a pas changé
 */
Console.WriteLine("{0} {1} {2}", p.Identifiant, p.Nom, p.Prenom);
```

```
c0058650-cdf9-4153-b4f1-f76004589cc7 TOURNESOL Tryphon
Le nom ne peut pas être vide ou null
c0058650-cdf9-4153-b4f1-f76004589cc7 TOURNESOL Tryphon
```

Méthodes publiques

```
/// <summary>
///
/// </summary>
/// <returns></returns>
public String affichage()
{
    return this.Nom + " - " + this.Prenom;
}
```


Surcharge de méthodes / polymorphisme

- **RI 169**
- Le polymorphisme paramétrique
  - Définition : c'est le fait d'avoir dans une même classe des méthodes de même nom mais avec un nombre différent de paramètres ou des types de paramètres différents.

```
/// <summary>
///
/// </summary>
/// <returns></returns>
public String affichage()
{
    return this.Nom + " - " + this.Prenom;
}

/// <summary>
/// méthode surchargée
/// Proposée une autre façon d'afficher...
/// </summary>
/// <param name="avecAge"></param>
/// <returns></returns>
public String affichage(Boolean avecAge)
{
    return this.affichage() + (avecAge ? String.Format(", {0} ans", this.Age) : String.Empty);

}
```

```
Console.WriteLine(p.affichage());
Console.WriteLine(p.affichage(true));
```

Membres partagés

- **RI 176 : le mot clé static**

- Ajoutez une propriétés Age qui retourne un entier (nombre d'années complètes entre 2 dates)

- Pour cela, créez une méthode dans le projet « Outils »:
  - AnneeDiff(d1 : Date, d2 : Date) : Integer

- Utilisation :

```
public int Age
{
    get { return ModOutils.AnneeDiff(this.DateNaissance, DateTime.Now); }
}
```


Méthodes d'extension

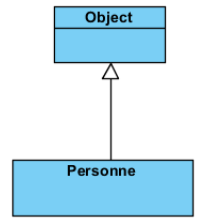
- **RI 173**
- Consigne :
  - Ecrivez le méthode PremiereLettreEnMajuscule(valeur:String, separateur:char)
    - Comportement: retourne la « valeur » avec la première lettre et la lettre suivant le séparateur en majuscule. Si la valeur est vide ou nulle, renvoie vide
    - Exemple:
      - jEan-baptiste => Jean-Batiste
  - « Branchez » son appel dans la propriété Prenom de la classe Personne

```
public String Prenom
{
    get { return _prenom; }
    set {
        /*
         * appel à la méthode d'extension
         */
        _prenom = value.premiereLettreEnMajuscule('-');
    }
}
```

RACKHAM – Le-Rouge

Héritage implicite

- La classe de base dans le Framework .NET est la classe **Object**.
- Si une classe n'hérite explicitement d'aucune classe, elle hérite implicitement de la classe Object.
- La classe Personne hérite donc des méthodes de la classe Object
  - Ex. : la méthode ToString()



Substitution de méthodes / polymorphisme

- **RI 170**
- La substitution de méthodes permet de redéfinir une méthode dans une classe dérivée.
  - Conditions nécessaires :
    - La méthode substituable doit avoir le modificateur **virtual**
    - La méthode de substitution doit avoir le modificateur **override**
    - La méthode de substitution doit avoir exactement la même signature que la méthode à substituer

public virtual string **ToString**()
Membre de System.Object

```
/// <summary>
/// méthode substituée
/// Spécialiser le comportement de ToString à la classe Personne
/// </summary>
/// <returns></returns>
public override string ToString()
{
    /*
     * on complete l'action de la méthode ToString de la classe Object
     */
    //return base.ToString() + " : " + this.affichage();
    /*
     * on remplace l'action de la méthode ToString de la classe Object
     */
    return this.affichage();
}
```

Méthode substituable

```
/// <summary>
/// méthode surchargée et substituable
/// Proposée une autre façon d'afficher...
/// </summary>
/// <param name="avecAge"></param>
/// <returns></returns>
public virtual String affichage(Boolean avecAge)
{
    return this.affichage() + (avecAge ? String.Format(", {0} ans", this.Age) : String.Empty);

}
```


Classe abstraite

- **RI 184**
- Définition
  - Une classe abstraite est une classe comme une autre mais qu'on ne peut pas instancier.
  - Une classe abstraite est souvent une classe racine d'une hiérarchie de classes.
  - Une classe est abstraite lorsqu'elle ne représente pas un concept réel manipulé dans une application.
    - Exemple: l'application *Gestion des employés* permet de gérer des employés. Elle ne va pas gérer des personnes mais seulement des employés. On peut donc dire que la classe Personne est une classe candidate pour être abstraite.

```
/// <summary>
/// Cette classe represente les informations à gér
/// </summary>
/// <remarks>Cette classe est abstraite</remarks>
public abstract class Personne
{
```

```
p = new Personne();
```

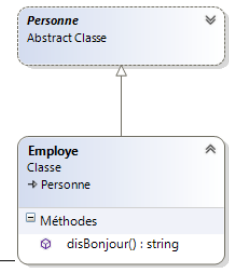
Méthodes abstraites

- Définir un service devant être rendu obligatoirement par les classes filles.

```
/// <summary>
/// Méhode abstraite
/// Définit le service de "comment dire Bonjour"
/// </summary>
/// <returns></returns>
public abstract String disBonjour();
```


Héritage explicite

- **RI 181**
- Représentation UML :



- Représentation C# :

```
/// <summary>
/// Classe spécialisant un individu en employé
/// </summary>
public class Employe : Personne
{
}
```

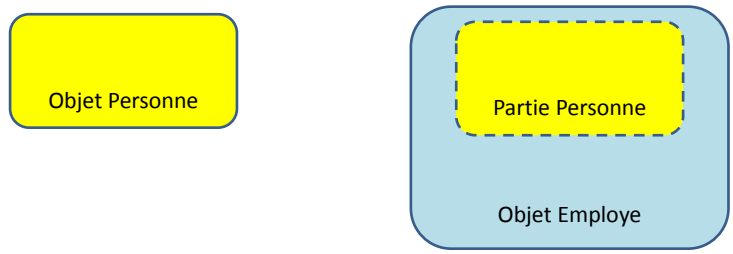
- Utilisation :

```
private static void testEmploye()
{
    Employe e = new Employe();
    Console.WriteLine(e.disBonjour());
}
```

```
Bonjour, je suis un employé. Mon nom n'est pas encore renseigné.
```

Représentation mémoire d'un objet

- Représentation d'un objet Personne et d'un objet Employe en mémoire :



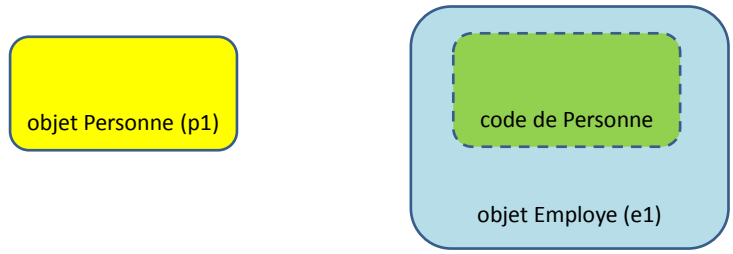
- Un objet Employe est un objet Personne avec une surcouche
- La construction d'un objet Employe utilise le code de construction d'un objet Personne.
- L'objet Employe dispose des attributs et comportements d'une Personne. Il peut également les spécialiser, les compléter.

Exercices

- A partir du diagramme de classes fourni, compléter la couche BO.

Transtypage d'objet / concept

- Le transtypage consiste à changer le point de vue par lequel on regarde un objet placé en mémoire.
  - Reprenons notre exemple: en mémoire, on a un objet Personne et un objet Employe



  - **p1** peut seulement s'envisager comme un élément de type Personne.
  - **e1** peut s'envisager comme une élément de type **Employe** mais aussi comme un élément de type **Personne** (il en embarque le code).
  - Quand on considère **e1** comme un élément de type Personne, on n'accède qu'aux seuls membres définis par le code de Personne dans l'objet e1.

Transtypage d'objet / Mise en oeuvre

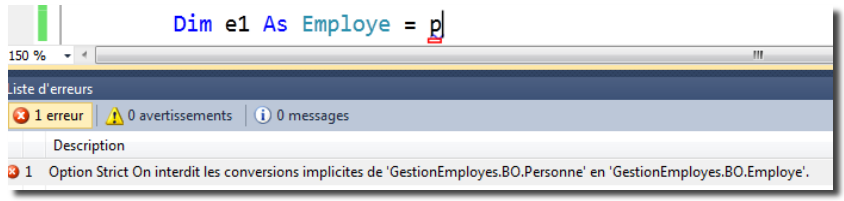
```
Employe e = new Employe(Guid.NewGuid(),
                        "Rastapopoulos",
                        "Roberto",
                        DateTime.Parse("04/12/1953"),
                        2500.25, DateTime.Parse("14/10/2000"));
//appel aux méthodes définies dans employe et personne
Console.WriteLine(e.affichage());
```

- Transtypage « ascendant » implicite

```
//Transtypage ascendant (implicite)
Personne p = e;
//appel aux méthodes définies dans personne et substituées dans employe
Console.WriteLine(p.affichage());
```

Transtypage d'objet / Mise en oeuvre

- Transtypage « descendant » explicite




```
//Transtypage descendant (explicite)
if (personnes[0].GetType().Equals(typeof(Employe)))
    ((Employe)personnes[0]).affichage(OptionsAffichageEmploye.Tout);

if (personnes[0] is Personne)
    ((Personne)personnes[0]).affichage();

Employe e2 = personnes[0] as Employe;
if (e2 != null)
    e2.affichage();
```

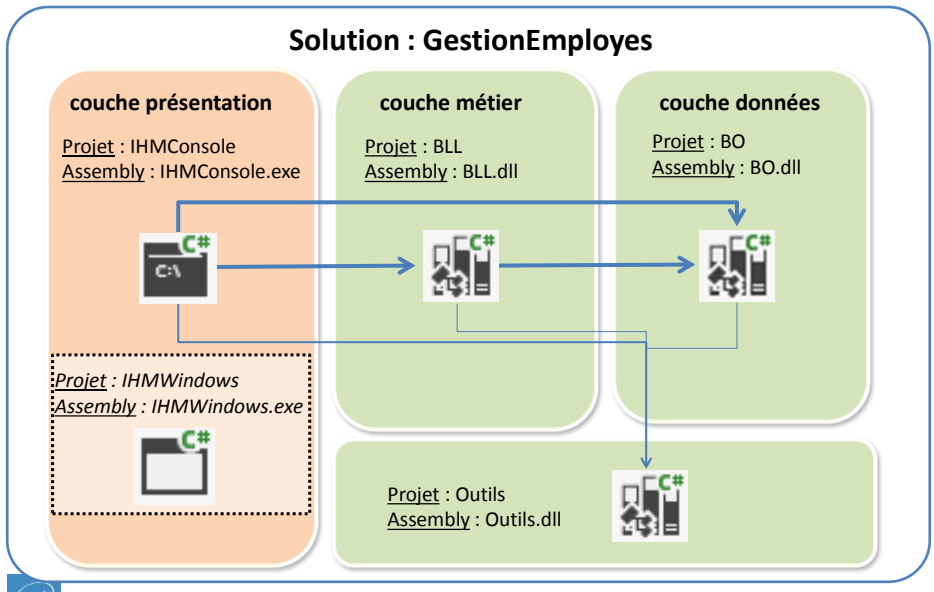

Développement en couches

Module 3

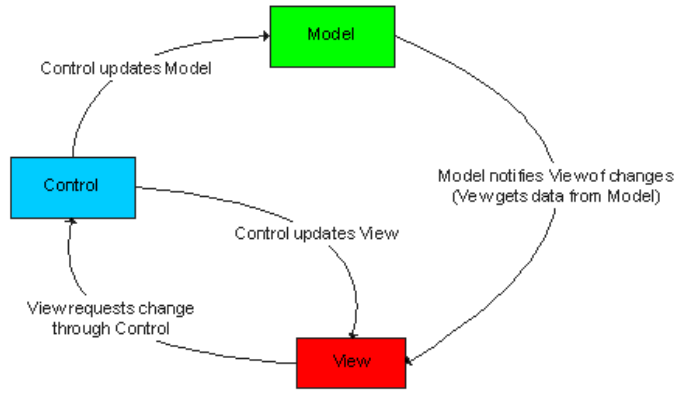
Développement de la couche BLL

Compléments POO avec C#

Architecture retenue : rappel

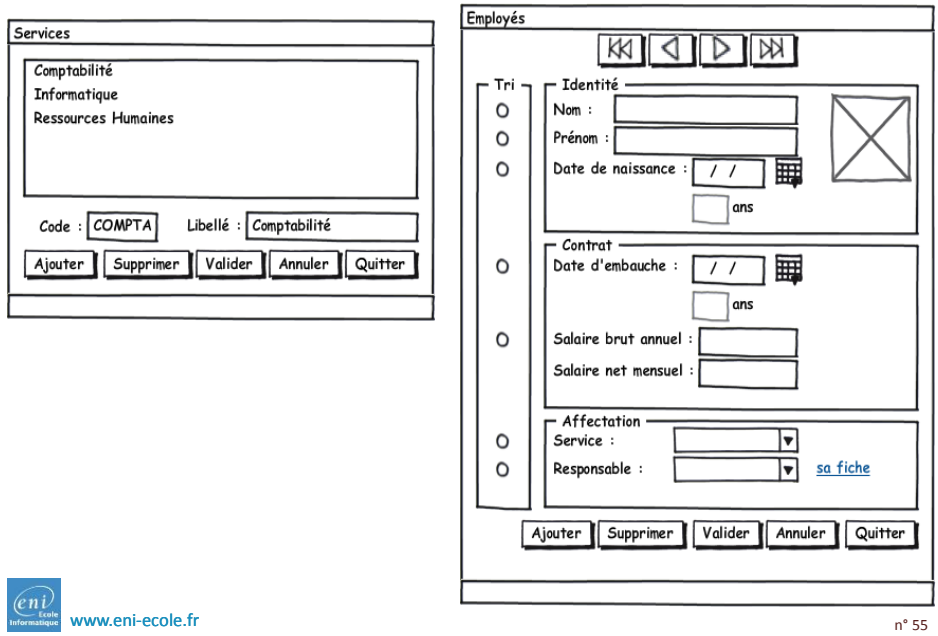


Le contrôleur



- Réagit aux demandes de la vue
- Met à jour le modèle
- Peut réclamer la mise à jour de la vue

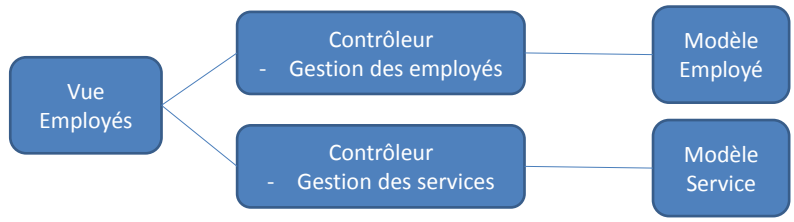
Définir les responsabilités des contrôleurs



Différentes philosophies

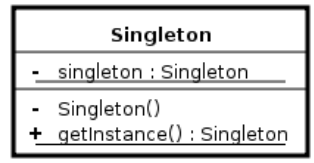
- Contrôleur fortement lié à la vue

- Contrôleur fortement lié au modèle (retenue pour le cours)



Design Pattern - Singleton

- Diagramme de classe

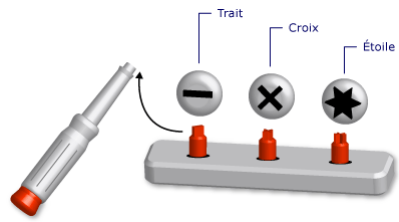


- Implémentation

```
public class Singleton
{
    /// <summary>
    /// stocker l'instance d'un objet
    /// </summary>
    private static Singleton _instance;
    /// <summary>
    /// constructeur privé
    /// </summary>
    private Singleton()
    {
    }
    /// <summary>
    /// Methode fournissant une instance de Singleton
    /// </summary>
    public static Singleton getInstance()
    {
        if (_instance == null)
        {
            _instance = new Singleton();
        }
        return _instance;
    }
}
```


Les types génériques 1/5

- **RI 198**
- Un type générique est un élément de programmation unique qui s'adapte pour exécuter les mêmes fonctionnalités pour différents types de données. Lorsque vous définissez une classe ou une procédure générique, vous ne devez pas définir une version distincte pour chaque type de données pour lequel vous souhaitez exécuter ces fonctionnalités.
- On pourrait comparer cet élément de programmation à un tournevis à têtes amovibles interchangeables. Vous pouvez examiner la vis que vous devez serrer et sélectionner la tête la mieux adaptée (fendue, en croix, en étoile). Quelle que soit la tête insérée dans le manche du tournevis, vous exécutez toujours la même fonction : vous serrez une vis.



Les types génériques 2/5

- Exemple d'utilisation du type List<T>

```
List<Service> services = new List<Service>();
services.Add(
```

```
void List<Service>.Add(Service item)
Ajoute un objet à la fin de System.Collections.Generic.List<T>.
item: Objet à ajouter à la fin de System.Collections.Generic.List<T>. La valeur peut être null pour les types référence.
```

- Les alternatives à un type générique sont :
  - un type unique utilisant le type de données Object ;
  - un ensemble de versions spécifiques au type du type, chaque version étant codée individuellement et utilisant un type de données spécifique tel que string, int ou un type défini par l'utilisateur tel que Service.

Les types génériques 3/5

- Avantages des types génériques par rapport aux alternatives
  - Sécurité de type
    - Vérification du type au moment de la compilation.
  - Performances
    - Les types génériques n'ont pas à effectuer des conversions.
  - Consolidation du code
    - Le code d'un type générique est défini une seule fois.
  - Réutilisation du code
    - Le code ne dépend pas d'un type générique. Il peut donc être réutilisé pour un nouveau type de données qui n'était pas prévu (dans certaines limites – Cf. diapo suivante).
  - Prise en charge IDE
    - IntelliSense prend en charge l'auto-complétion des types génériques.
  - Algorithmes génériques

Les types générique 4/5

- Le vocabulaire
  - **Type générique (*Generic Type*)**
    - Définition d'une classe, structure, interface, procédure ou délégué pour laquelle vous fournissez au moins un type de données lors de sa déclaration.
  - **Paramètre de type (*Type Parameter*)**
    - Dans une définition de type générique, espace réservé pour un type de données fourni lors de la déclaration du type.
  - **Argument de type (*Type Argument*)**
    - Type de données spécifique qui remplace un paramètre de type lorsque vous déclarez un type construit à partir d'un type générique.
  - **Contrainte (*Constraint*)**
    - Condition sur un paramètre de type qui restreint l'argument de type que vous pouvez lui fournir.
  - **Type construit (*Constructed Type*)**
    - Classe, structure, interface, procédure ou délégué déclaré à partir d'un type générique en fournissant des arguments de type pour ses paramètres de type.

Les types génériques 5/5

- Deux déclarations de types génériques
  - Un type générique simple
    - **Type** peut être n'importe quel type

```
/// <summary>
/// classe générique simple
/// </summary>
/// <typeparam name="Type"></typeparam>
public class ClasseGenerique1<Type>
{
    private Type _attribut;
    public void methode(Type arg)
    {

    }
}
```

  - Un type générique avec des contraintes
    - **Type** doit être du type Voiture (ou dérivé) ***et*** implémenter l'interface ICotation.

```
/// <summary>
/// classe générique n'acceptant que des sortes de voitures cotables
/// </summary>
/// <typeparam name="Type"></typeparam>
public class ClasseGenerique4<Type> where Type : Voiture, ICotation
{
    private Type _attribut;
    public void methode(Type arg)
    {

    }
}
```

Exercices

- A partir du diagramme de classes fourni, compléter la classe MgtService

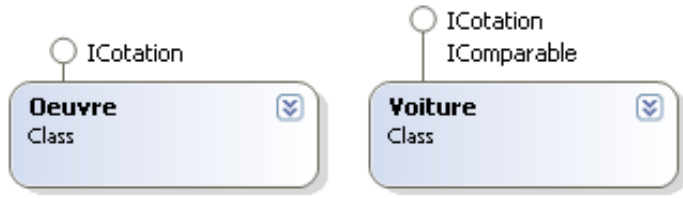
Interface 1/4

- **RI 187**
- Définition
  - Une interface permet de définir des comportements.
  - L'interface peut être associée à toute classe souhaitant bénéficier de ces comportements.
  - La classe doit obligatoirement redéfinir les comportements proposés par l'interface.
- Déclaration

```
/// <summary>
/// gestion d'un service de cotation
/// </summary>
public interface ICotation
{
    /// <summary>
    /// Stocker la cote d'un objet
    /// </summary>
    /// <remarks>L'objet doit être en mesure de stocker sa tendance</remarks>
    /// <example private bool _tendance;/>
    decimal Cote { get; set; }
    /// <summary>
    /// retourne la tendance de l'objet
    /// </summary>
    /// <returns></returns>
    bool isEnHAusse();
}
```

Interface 2/4

- Utilisation



Interface 3/4

```
/// <summary>
/// Une oeuvre utilise également le service de cotation
/// </summary>
public class Oeuvre
    :ICotation
{
    #region "membres issus de l'interface ICotation"
    private decimal _cote;
    private bool _tendance;
    public decimal Cote
    {
        get
        {
            return _cote;
        }
        set
        {
            if (value < this._cote)
                this._tendance = false;
            else this._tendance = true;
            this._cote = value;
        }
    }

    public bool isEnHAusse()
    {
        return this._tendance;
    }
    #endregion
```

Interface 4/4

- Un exemple au travers du Framework

List<T>.Sort, méthode

Notes

Cette méthode utilise le comparateur par défaut Comparer<T>.Default pour le type *T* afin de déterminer l'ordre des éléments de liste. La propriété Comparer<T>.Default vérifie si le type *T* implémente l'interface générique IComparable<T> et utilise cette implémentation, le cas échéant. Sinon, Comparer<T>.Default vérifie si le type *T* implémente l'interface IComparable. Si le type *T* n'implémente aucune des interfaces, Comparer<T>.Default lève InvalidOperationException.

```
public int CompareTo(Service other)
{
    return this.Code.ToUpper().CompareTo(other.Code.ToUpper());
}
#endregion
```


Délégués 1/4

- **RI 194**
- Définition
  - Un délégué est une classe pouvant contenir une référence à une méthode. A la différence d'autres classes, une classe déléguée possède une signature et elle ne peut contenir que des références à des méthodes qui correspondent à sa signature.
- Déclaration

```
private delegate String DisBonjour(String texte);
```

- Instanciation
  - On va pouvoir créer une instance de la classe déléguée en lui passant en paramètre une méthode correspondant à la signature définie

```
DisBonjour bonjour = sayHello;
afficheMessageSalutation(bonjour, "Jean");
```

Délégués 2/4

- Les méthodes passées en paramètre du constructeur du délégué doivent respecter la signature définie

```
// déclaration de la méthode à passer en paramètre respectant la signature du délégué
private static String disBonjour(String nom) {
    return "Bonjour " + nom + " !";
}

// déclaration d'une autre méthode à passer en paramètre respectant la signature du délégué
private static String sayHello(String nom) {
    return "Hello " + nom + " !";
}
```

- Exécution (avec ou sans la méthode **Invoke** du délégué)

```
// méthode prenant en paramètre un délégué
private static void afficheMessageSalutation(DisBonjour d, String param)
{
    Console.WriteLine(d.Invoke(param));
    Console.WriteLine(d(param));
}
```


Délégués 3/4

- Intérêts
  - Passer une fonction en paramètre d'une autre fonction afin de spécialiser son comportement.
  - Exemple :
    - Soit une liste de services que l'on veut trier par libellé.
    - List(Of T) possède l'algorithme de tri au travers des méthodes **Sort**

Syntaxe

C# C++ F# VB

```
public void Sort(
        Comparison<T> comparison
)
```

    - Nous devons l'informer de l'algorithme de comparaison à utiliser.
    - Une solution est de lui passer la méthode de comparaison respectant la signature du délégué

Syntaxe

C# C++ F# VB

```
public delegate int Comparison<in T>(
        T x,
        T y
)
```

Délégués 4/4

- Réalisation du tri en passant à la méthode **Sort** la méthode de comparaison :

```
Comparison<Service> delegue = null;
delegue = TriSurLibelle;
listeTemp.Sort(delegue);
```

- La méthode de tri sur le libellé respectant la signature du délégué :

```
private int TriSurLibelle(Service s1, Service s2)
{
    return s1.Libelle.ToUpper().CompareTo(s2.Libelle.ToUpper());
}
```


Expressions lambda

- **RI 196**
- Une expression lambda est une fonction sans nom qui peut être utilisée à la place d'un délégué.
- Une expression lambda commence par la déclaration de la signature du délégué
- Exemple d'une expression lambda écrite sur une ligne :

```
_listeServices.ToList().Find((Service s) => s.Code.ToUpper().Equals(code.ToUpper()));
```

- Exemple d'une expression lambda sur plusieurs ligne :

```
listeTemp.Sort((Service s1, Service s2) => {
    return s1.Code.ToUpper().CompareTo(s2.Code.ToUpper());
    });
```

Exercices

- compléter la classe MgtService :
  - Mise en place du énumération exposant les 3 critères de tri de la liste des services
    - Par défaut (code), Libelle, Code
  - Proposer une méthode de tri permettant d'utiliser :
    - L'interface IComparable pour le tri par défaut,
    - Un délégué pour le tri sur le libellé,
    - Une expression lamdba pour le tri sur le code.
  - Utiliser les expressions lambda dans les tests des méthodes CRUD.

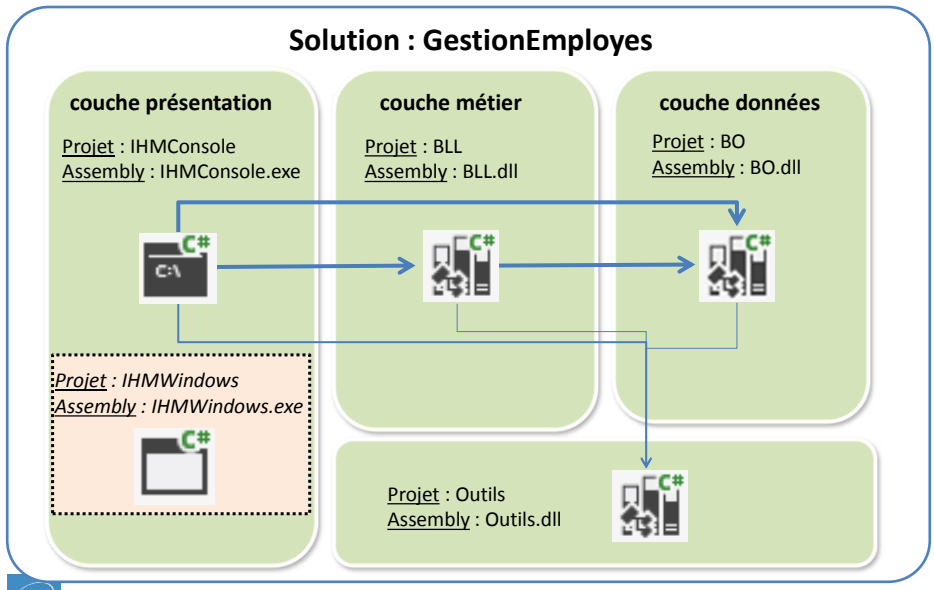
Développement en couches

Module 4

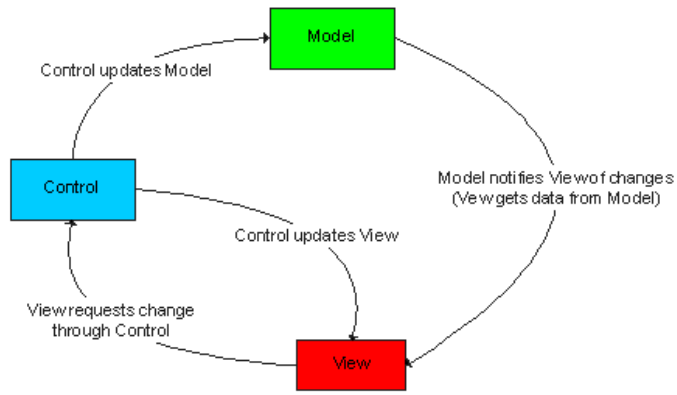
Développement de la couche IHM

Compléments POO avec C#

Architecture retenue : rappel



La vue



- Intercepte toutes les actions de l'utilisateur
- Délègue le traitement au contrôleur
- Présente les données à l'utilisateur

Exercice

- Mise en place de l'IHM de gestion des services en mode console.

Evénements 1/4

- **RI 191**
- Définition:
  - Un événement est un signal qui informe l'application que quelque-chose d'important s'est produit.
  - Exemple:
    - Lorsqu'un utilisateur clique sur le contrôle d'un formulaire, le formulaire peut déclencher un événement Click et appeler une procédure qui gère l'événement.
    - Les événements séparent également les tâches pour communiquer. Par exemple, votre application exécute une tâche de tri séparément de l'application principale. Si un utilisateur annule le tri, votre application peut envoyer un événement d'annulation demandant au processus de tri de s'interrompre.

Evénements 2/4

- Termes utilisées
  - Déclaration d'événements

```
public event EventHandler ListeTriee;
public event EventHandler<ListeTrieeEventArgs> ListeTrieeSur;
```

    - EventHandler : délégué représentant la méthode qui gèrera un évènement.

  - Déclenchement d'événements
    - Propager l'évènement si un gestionnaire est à l'écoute.

```
protected virtual void onListeTriee(EventArgs e)
{
    EventHandler handler = ListeTriee;
    if (handler != null) handler(this, e);
}
protected virtual void onListeTrieeSur(ListeTrieeEventArgs e)
{
    EventHandler<ListeTrieeEventArgs> handler = ListeTrieeSur;
    if (handler != null) handler(this, e);
}
```


Evénements 3/4

    - Déclencher l'évènement au bon moment.

```
onListeTriee(EventArgs.Empty);

onListeTrieeSur(new ListeTrieeEventArgs(CritereService.Code));
```

  - Gestionnaires d'événements (*Ecouteur*)
    - Le gestionnaire d'un événement est la méthode exécutée quand un événement est déclenché. Sa signature correspond à celle du délégué.
    - Il est possible d'avoir un gestionnaire d'évènement pour plusieurs évènements déclenchés.
    - Il est possible d'avoir plusieurs gestionnaires d'événements pour le même événement déclenché. Ils seront appelés les uns après les autres dans l'ordre de l'association (+=)
    - Cette méthode doit respecter la signature du Handler associé à l'évènement lors de sa déclaration

```
public delegate void EventHandler(
        Object sender,
        EventArgs e
)
```

Evénements 4/4

```
private void gestion_ListeTriee(Object sender, EventArgs e)
{
    _Message = "Liste des services triée par défaut";
}
private void gestion_ListeTrieeSur(Object sender, ListeTrieeEventArgs e)
{
    _Message = "Liste des services triée (gestion addHandler)" +
        " sur " + e.Critere + " a " + e.Heure;
}
```

- Association d'événements aux gestionnaires d'événements :

```
gestion.ListeTriee += gestion_ListeTriee;
gestion.ListeTrieeSur += gestion_ListeTrieeSur;
```


BindingList<T> et INotifyPropertyChanged

- Une BindingList<T> notifie des ajouts et suppressions d'objets dont elle a la responsabilité.
  - Prise en charge de l'évènement ListChanged

```
gestion.ServicesList.ListChanged += gestion_ListeChangee;

/// <summary>
/// Ecouteur de l'évènement levé par la BindingList
/// </summary>
/// <param name="sender"></param>
/// <param name="e"></param>
private void gestion_ListeChangee(object sender, System.ComponentModel.ListChangedEventArgs e)
{
    switch (e.ListChangedType)
    {
        case System.ComponentModel.ListChangedType.ItemAdded:
            _Message = "nouveau service ajouté";
            break;
        case System.ComponentModel.ListChangedType.ItemDeleted:
            _Message = "service supprimé";
            break;
    }

}
```

BindingList<T> et INotifyPropertyChanged

- Problème : la modification d'un objet géré par BindingList ne déclenche pas l'évènement ListChanged par défaut.
- Solution : une classe peut notifier des modifications apportées à ses propriétés.
  - interface **INotifyPropertyChanged** de l'espace de nom System.ComponentModel.

```
public class Service : IComparable<Service>, INotifyPropertyChanged
```

  - Implémenter l'interface et définir le déclencheur.

```
#region "Implémentation de l'interface INotifyPropertyChanged"
public event PropertyChangedEventHandler PropertyChanged;

/// <summary>
/// méthode de déclenchement de l'évènement
/// </summary>
/// <param name="propertyName"></param>
protected void OnPropertyChanged(string propertyName)
{
    PropertyChangedEventHandler handler = PropertyChanged;
    if (handler != null)
    {
        handler(this, new PropertyChangedEventArgs(propertyName));
    }
}
#endregion
```


BindingList<T> et INotifyPropertyChanged

- Lever l'évènement lors de la modification d'une propriété

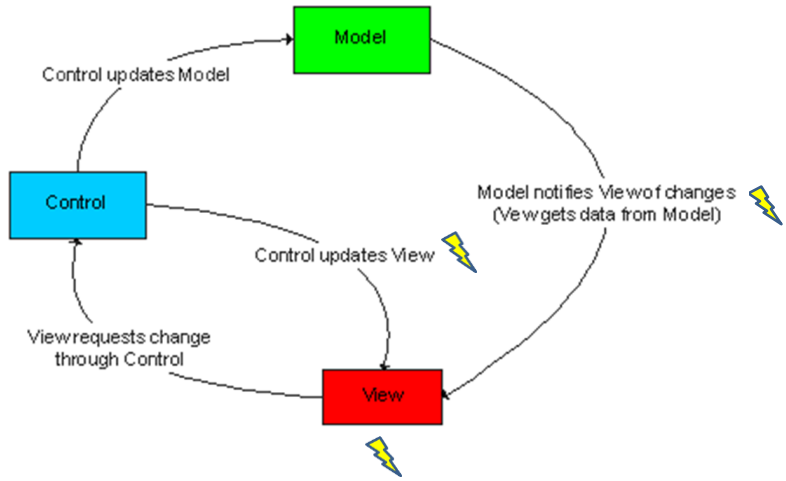
```
public String Code
{
    get { return _code; }
    set {
        verifCode(value);
        _code = value.ToUpper();
        /*
         * notifier du changement d'état de la propriété Code
         */
        OnPropertyChanged("Code");
    }
}
```

BindingList<T> et INotifyPropertyChanged

- BindingList déclenche maintenant l'évènement ListChanged sur la modification d'un objet.

```
/// <summary>
/// Ecouteur de l'évènement levé par la BindingList
/// </summary>
/// <param name="sender"></param>
/// <param name="e"></param>
private void gestion_ListeChangee(object sender, System.ComponentModel.ListChangedEventArgs e)
{
    switch (e.ListChangedType)
    {
        case System.ComponentModel.ListChangedType.ItemAdded:
            _Message = "nouveau service ajouté";
            break;
        case System.ComponentModel.ListChangedType.ItemDeleted:
            _Message = "service supprimé";
            break;
        case System.ComponentModel.ListChangedType.ItemChanged:
            _Message = "service modifié";
            break;
    }

}
```


Evènements dans le modèle MVC

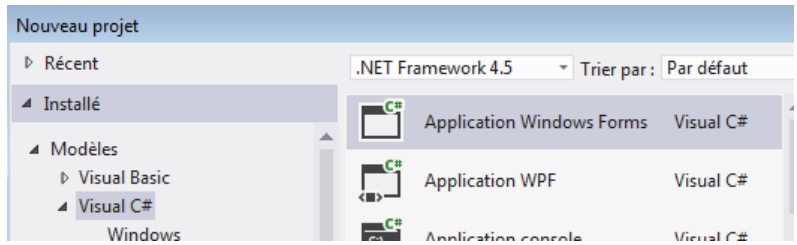


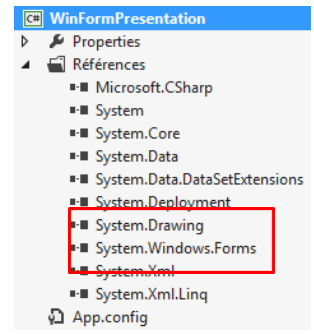
Développement en couches

Module 5

Développement de la couche IHM : composants Windows Forms

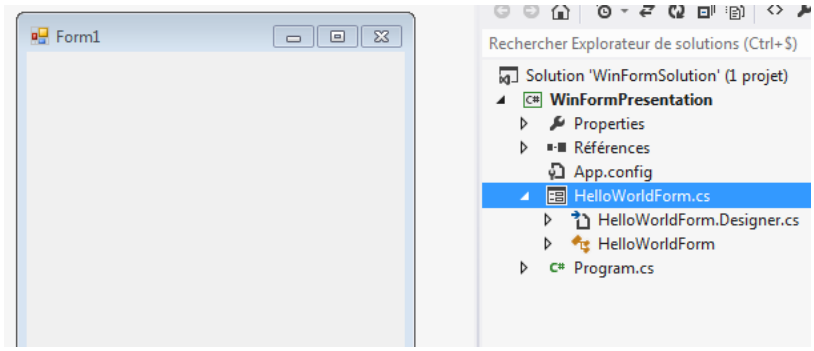
Le projet Windows Forms





Structure du formulaire

- **RI 252**
- L'éditeur graphique
- Les fichiers :
  - HelloWorldForm.cs
  - HelloWorldForm.Designer.cs



Structure du formulaire

- Héritage de la classe Form
- Classe partielle

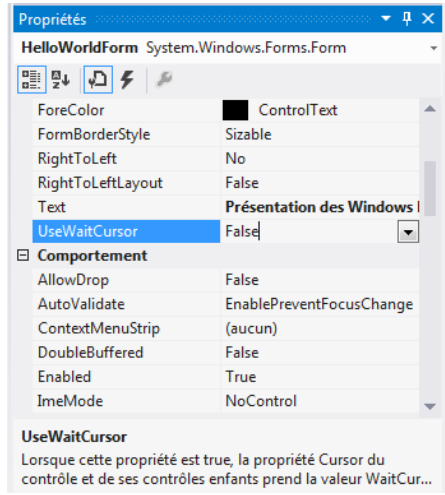
```
public partial class HelloWorldForm : Form
{
    public HelloWorldForm()
    {
        InitializeComponent();
    }
}
```

- La méthode InitializeComponent()

```
/// <summary>
/// Méthode requise pour la prise en charge du concepteur - ne modifiez pas
/// le contenu de cette méthode avec l'éditeur de code.
/// </summary>
private void InitializeComponent()
{
    this.components = new System.ComponentModel.Container();
    this.AutoScaleMode = System.Windows.Forms.AutoScaleMode.Font;
    this.Text = "Form1";
}
```

Personnaliser le formulaire

- **RI 254-260**
- A partir de la palette de propriétés (raccourci – F4) proposée par le concepteur de vue



Personnaliser le formulaire

- La méthode InitializeComponent() est automatiquement mise à jour.

```
private void InitializeComponent()
{
    this.SuspendLayout();
    //
    // HelloWorldForm
    //
    this.AutoScaleDimensions = new System.Drawing.SizeF(6F, 13F);
    this.AutoScaleMode = System.Windows.Forms.AutoScaleMode.Font;
    this.BackColor = System.Drawing.Color.White;
    this.ClientSize = new System.Drawing.Size(484, 401);
    this.ForeColor = System.Drawing.Color.FromArgb(((int)(((byte)(
    this.Location = new System.Drawing.Point(100, 100);
    this.MaximumSize = new System.Drawing.Size(500, 500);
    this.MinimizeBox = false;
    this.Name = "HelloWorldForm";
    this.Text = "Présentation des Windows Forms";
    this.ResumeLayout(false);

}
```

Définir le formulaire de démarrage

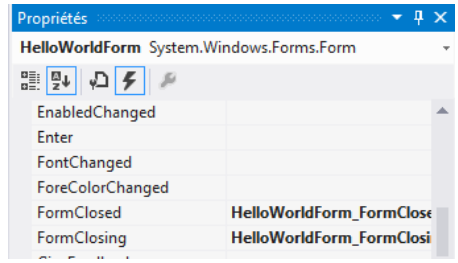
- Classe Program.cs.

```
static class Program
{
    /// <summary>
    /// Point d'entrée principal de l'application.
    /// </summary>
    [STAThread]
    static void Main()
    {
        Application.EnableVisualStyles();
        Application.SetCompatibleTextRenderingDefault(false);
        Application.Run(new HelloWorldForm());
    }
}
```

	Run(Form)	Commence à exécuter une boucle de messages d'application standard sur le thread en cours et affiche le formulaire spécifié.

Gérer les évènements du formulaire

- Mise en place du gestionnaire d'évènement à partir de la palette de propriétés



- L'évènement et son gestionnaire sont associés dans la méthode InitializeComponent()

```
this.FormClosing += new System.Windows.Forms.FormClosingEventHandler(this.HelloWorldForm_FormClosing);
this.FormClosed += new System.Windows.Forms.FormClosedEventHandler(this.HelloWorldForm_FormClosed);
```

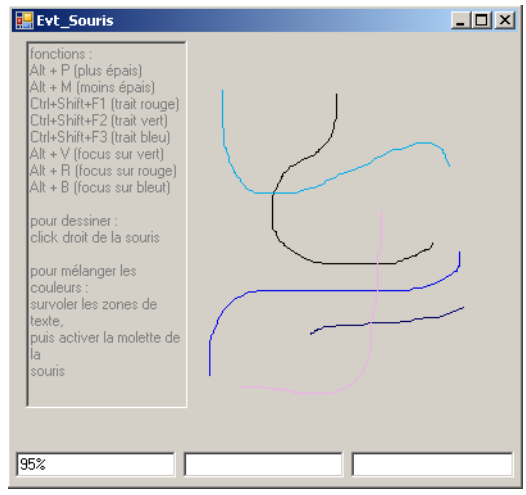
Gérer les évènements du formulaire

- Ecriture du gestionnaire d'évènement

```
private void HelloWorldForm_FormClosing(object sender, FormClosingEventArgs e)
{
    //
    //Interrompre l'évènement
    //
    {
        if (MessageBox.Show("Souhaitez fermer le formulaire ?", "Fermeture en cours",
            MessageBoxButtons.YesNo, MessageBoxIcon.Question,
            MessageBoxDefaultButton.Button2) == DialogResult.No)
        {
            e.Cancel = true;
        }
    }
}
```

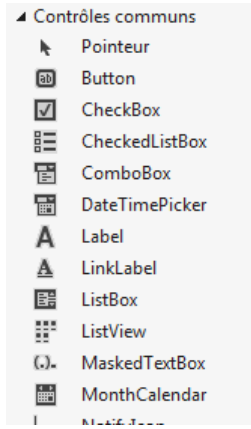

Exercices

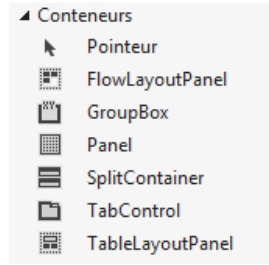
- TP Ardoise Magique
- Prise en main des évènements Clavier et Souris **RI 266**
- Utilisation d'un contrôle TextBox **RI 313**

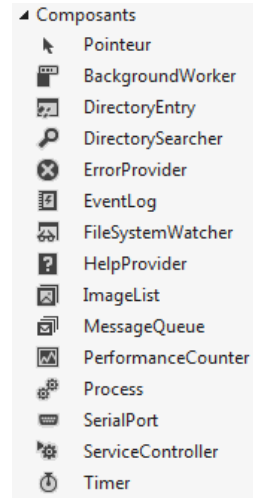


Utilisation des contrôles graphiques

- Différents types de composants
  - Contrôles communs
  - Conteneurs
  - Composants

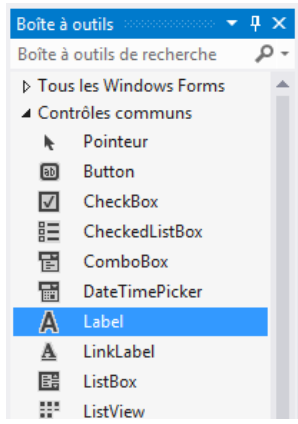




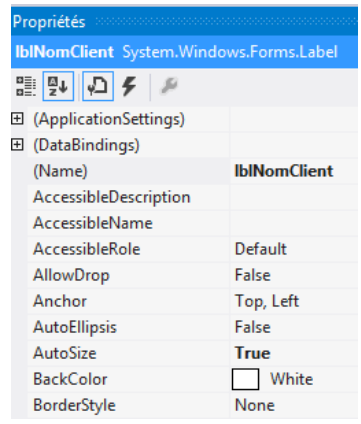


Utilisation des contrôles graphiques

- **RI 286**
- Ajout de contrôles



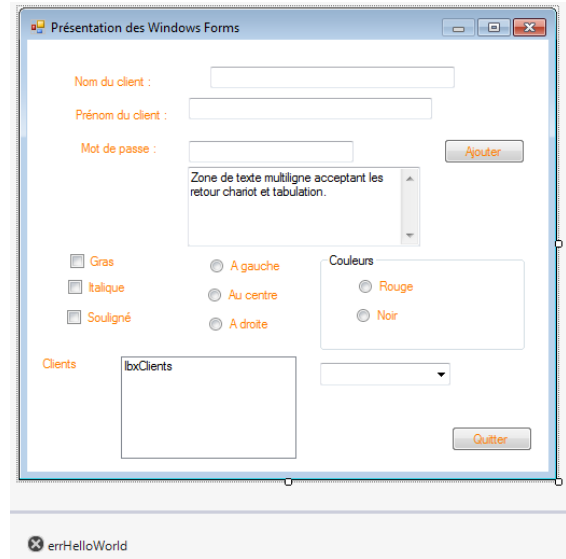
- Personnaliser les propriétés



- Propriétés et comportements de la classe Control **RI 297**

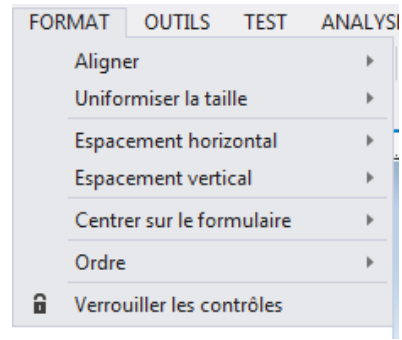
Utilisation des contrôles graphiques

- Label **RI 304**
- TextBox **RI 313**
- Button **RI 319**
- CheckBox **RI 326**
- RadioButton **RI 328**
- Panel & GroupBox **RI 342**
- ListBox **RI 330**
- ComboBox **RI 334**
- ErrorProvider **RI 309**
- ToolTip **RI 308**



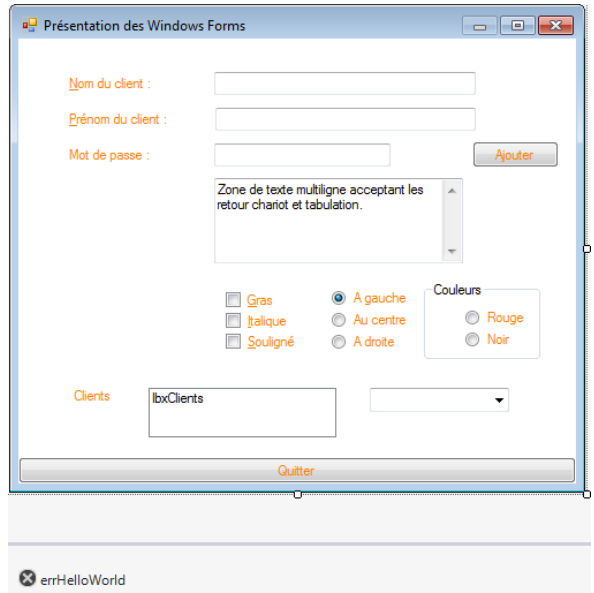
Ergonomie du formulaire

- **RI 288**
- Positionner et dimensionner les contrôles



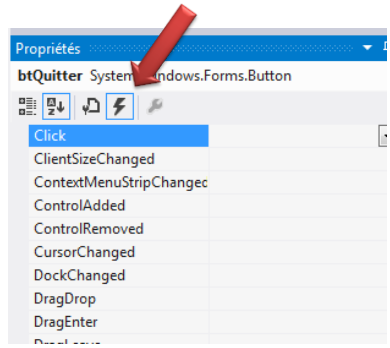
- Ancrage des contrôles **RI 292**
- Verrouiller les contrôles
- Passer le focus entre les contrôles **RI 294**
- Mise en place de raccourcis-clavier **RI 296**

Ergonomie du formulaire



Implémenter les gestionnaires d'évènements sur les contrôles

- Sélectionner le composant dans le concepteur de vue
- Accéder à la palette des propriétés – Evènements



Implémenter les gestionnaires d'évènements sur les contrôles

- Double cliquer sur l'évènement choisi
  - Le gestionnaire d'évènement est créé

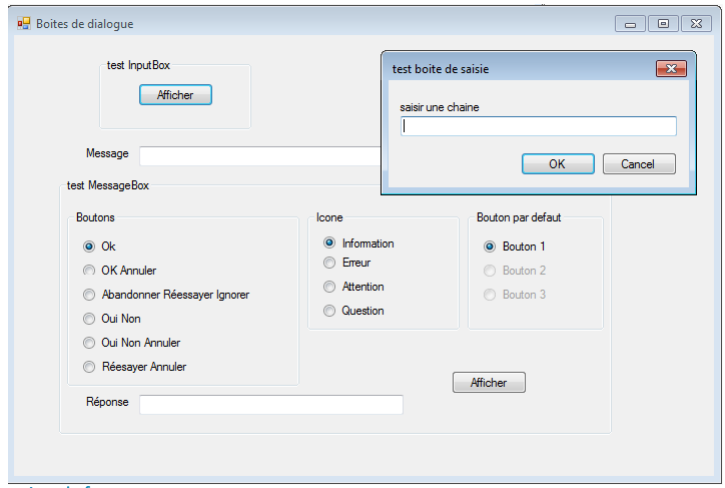
```
private void btQuitter_Click(object sender, EventArgs e)
{

}
```

  - L'évènement et son gestionnaire sont câblés.

```
this.btQuitter.Click += new System.EventHandler(this.btQuitter_Click);
```

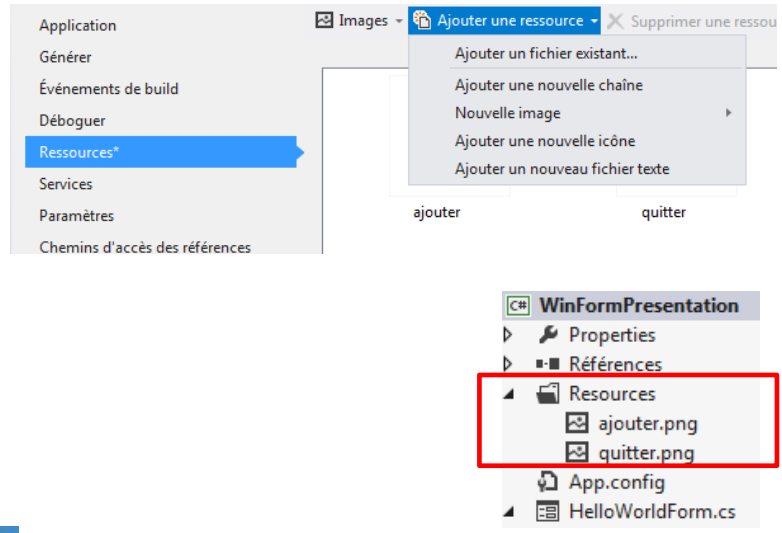

Exercices

- TP BoxForm
- Prise en main des boites de dialogue **RI 274**
- Utilisation des contrôles



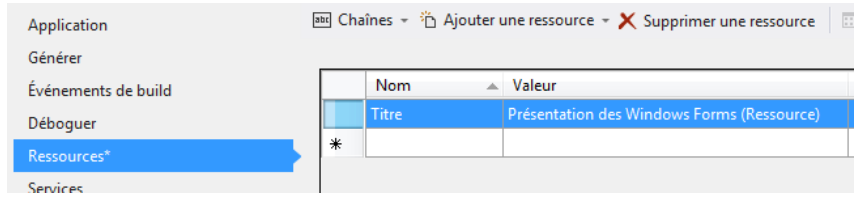
Ajouter des ressources au projet

- A partir des propriétés du projet



Ajouter des ressources au projet

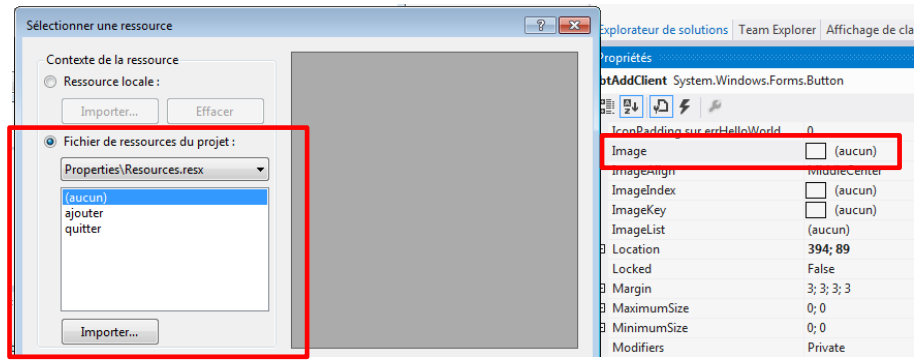
- A partir des propriétés du projet (suite)



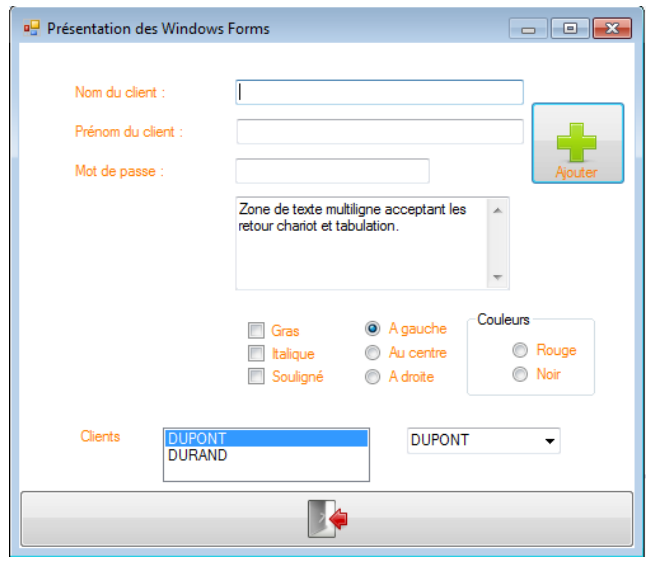

```
try
{
    this.Text = WinFormPresentation.Properties.Resources.Titre;
}
catch
{
    MessageBox.Show("La ressource ne peut être chargée !");
}
```

Ajouter des ressources au projet

- A partir des propriétés d'un contrôle



Ajouter des ressources au projet



La validation de la saisie

- Validation au niveau des champs
  - Les propriétés de validation
    - MaxLength, …
    - CauseValidation
  - Les évènements de validation
    - KeyDown, KeyUp, KeyPress
    - Validating, Validated
- Validation au niveau du formulaire
  - Le composant ErrorProvider

```
private void notifierErreur(Control c, string message)
{
    errHelloWorld.SetIconPadding(c, 5);
    errHelloWorld.SetIconAlignment(c, ErrorIconAlignment.MiddleRight);
    errHelloWorld.SetError(c, message);
}
```

  - Propriété et méthode du formulaire
    - AutoValidate, ValidateChildren(ValidationConstraint constraint).

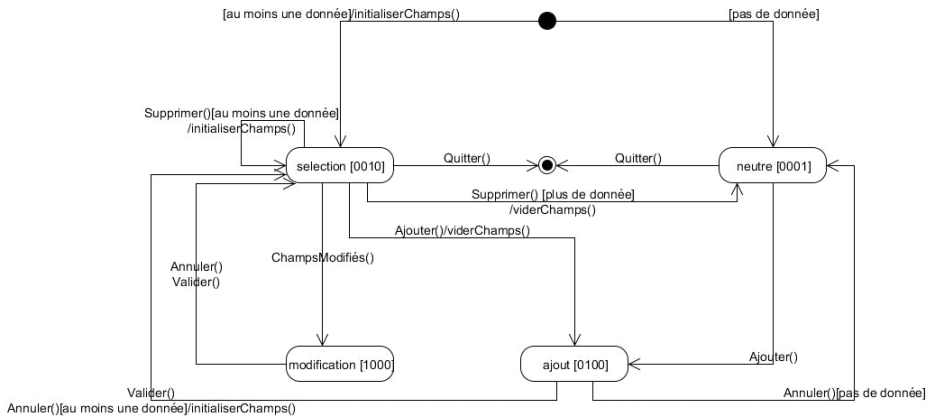
Gérer l'activation des contrôles

- Problème : activer/désactiver les contrôles en fonction de l'état du formulaire.
- Objectif : guider l'utilisateur dans son utilisation de l'écran.
- 4 états principaux associés à un masque binaire distinct :
  - Neutre : l'écran est vide
  - Sélection : un enregistrement est présenté sur l'écran
  - Ajout : l'utilisateur demande à créer un nouvel enregistrement
  - Modification : l'utilisateur modifie une donnée de l'enregistrement présenté sur l'écran.

```
/// <summary>
/// identifier les différents états associés à l'écran
/// </summary>
[Flags]
public enum ModesEcran
{
    Neutre = 1,      //0001
    Selection = 2,   //0010
    Ajout = 4,       //0100
    Modification = 8//1000
}
```

Gérer l'activation des contrôles

- Diagramme d'états-transitions du formulaire



Gérer l'activation des contrôles

- Associer à chaque contrôle les modes pour lesquels il doit s'activer.
  - Problème: Comment affecter deux valeurs à une seule propriété ?
  - L'utilisation des opérateurs binaires permet de résoudre ce problème
    - Le OU binaire

OU	0	1
0	0	1
1	1	1

- Exemple: 00010010(18) OU 00000111(7) = 00010111(23)

    - Le ET binaire

ET	0	1
0	0	0
1	0	1

- Exemple: 00010010(18) ET 00000111(7) = 00000010(2)

Gérer l'activation des contrôles

- Comment les utiliser dans notre contexte ?
  - Une valeur entière est associée explicitement à chaque élément de l'énumération ModesEcran :

Enum	Valeur entière	Valeur binaire
Neutre	1	00000001
Selection	2	00000010
Ajout	4	00000100
Modification	8	00001000

  - Les valeurs sont choisies pour faire en sorte qu'un seul bit soit initialisé à 1
  - La somme de valeurs de l'énumération est toujours différente de la valeur d'un élément de l'énumération.
  - Si on avait besoin de valeurs supplémentaires, on utiliserait 16, 32, 64…

Gérer l'activation des contrôles

- Affectation d'une valeur avec le OU binaire

```
/*
 * associer à chaque contrôle, les états qui l'activent
 */
btAjouter.Tag = ModesEcran.Neutre | ModesEcran.Selection;
btSupprimer.Tag = ModesEcran.Selection;
btAnnuler.Tag = ModesEcran.Ajout | ModesEcran.Modification;
btValider.Tag = ModesEcran.Ajout | ModesEcran.Modification;
btQuitter.Tag = ModesEcran.Neutre | ModesEcran.Selection;

tbData.Tag = ModesEcran.Selection | ModesEcran.Ajout | ModesEcran.Modification;
```

  - La valeur de Tag est alors

```
btAjouter.Tag = ModesEcran.Neutre | ModesEcran.Selection;
```

    - 00000001(1) OU 00000010(2) = 00000011(3)
    - Le bit de *Neutre* (1) et de *Selection* (2) sont positionnés à 1 dans la valeur finale (3).
    - On sait que 3 n'est possible que par l'association de *Neutre* (1) et de *Selection* (2) .
  - On a ainsi pu, dans une seule propriété, enregistrer deux informations.

Gérer l'activation des contrôles

- Lecture de la propriété **Tag** avec un ET binaire
  - Problème :
    - Si on lit la propriété Tag, on aura la valeur 12.
    - Cette valeur ne correspond à aucune valeur de l'énumération **ModesEcran.**
  - Solution :
    - Le ET binaire va nous permettre de décomposer cette valeur.

```
btAjouter.Enabled = (((ModesEcran)(btAjouter.Tag) & Mode) == Mode);
```

    - Quelle opération a été réalisée?
      - 00001100(12) ET 00000100(4)
      - Les valeurs possibles sont 00000000(0) / 00000100(4).
      - Elle nous permet de savoir si le bit de la valeur *Ajout* (4) est à 1.
    - Il faut faire ce test pour chacune des valeurs de l'énumération.

Gérer l'activation des contrôles

- Faire varier les modes en fonction de l'action réalisée.

```
private void btAjouter_Click(object sender, EventArgs e)
{
    Mode = ModesEcran.Ajout;
}

private void btSupprimer_Click(object sender, EventArgs e)
{
    //supprimer les données
    datas.RemoveAt(0);

    //puis...
    NeutreOuSelection();

}
```

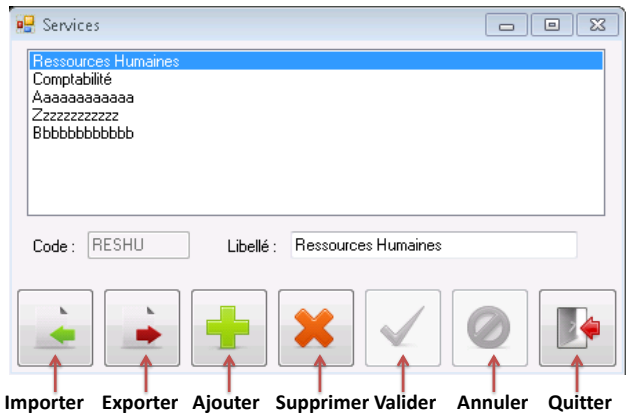
Gérer l'activation des contrôles

- Faire varier les modes en fonction de l'action réalisée.

```
private void tbData_Validating(object sender, CancelEventArgs e)
{
    if (String.IsNullOrWhiteSpace(tbData.Text))
        ControleSaisie.SetError(tbData, "La saisie est obligatoire.");
    else
        ControleSaisie.SetError(tbData, string.Empty);

}

private void tbData_KeyPress(object sender, KeyPressEventArgs e)
{
    if (Mode == ModesEcran.Selection)
        Mode = ModesEcran.Modification;

}
```

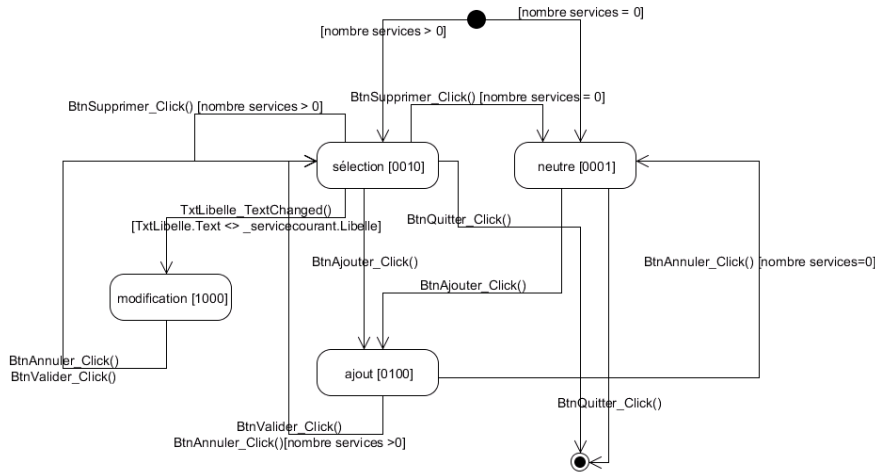

Exercices

- Ajoutez le projet IHMWindows, associez-le au namespace GestionEmployes.IHM. Ce projet devient le projet de démarrage de la solution.
- Référencez les projets BLL, BO et Outils auprès du projet IHMWindows.
- Créez la fenêtre **FrmServices** et définissez son interface graphique.



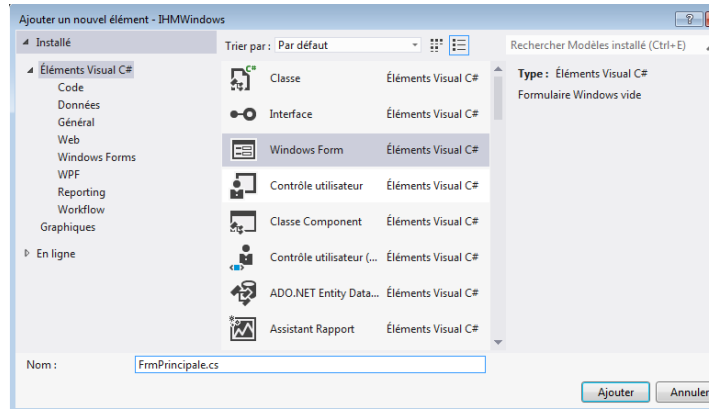
Exercices

- En vous appuyant sur la classe **MgtService**, mettez en place les scénarios suivants :



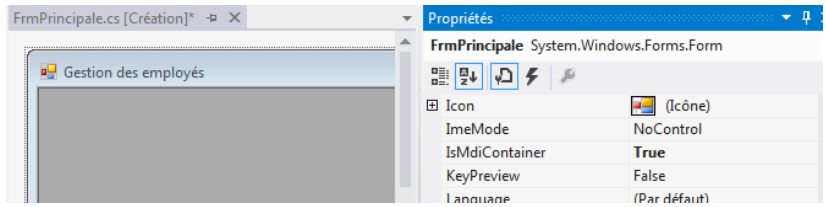
Applications MDI

- RI 260
- Elles sont constituées de deux types de feuilles :
  - La feuille mère
  - Les feuilles filles
- Pour ajouter une feuille mère :



Applications MDI

- Modifier la propriété **IsMdiContainer** à True

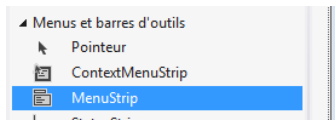


- La fenêtre MDI est définie comme fenêtre de démarrage.

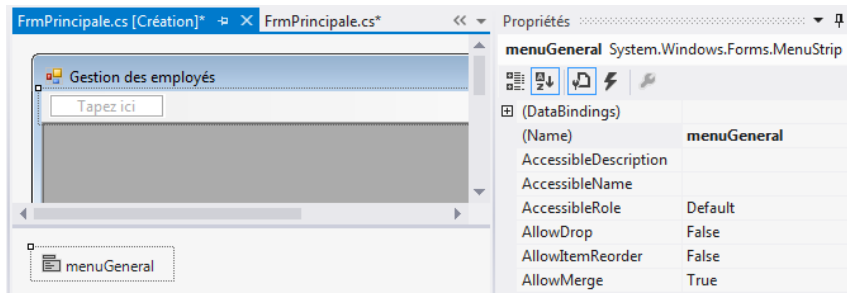
```
static void Main()
{
    Application.EnableVisualStyles();
    Application.SetCompatibleTextRenderingDefault(false);
    Application.Run(new FrmPrincipale());
}
```


Menu

- Le contrôle **MenuStrip** RI 319

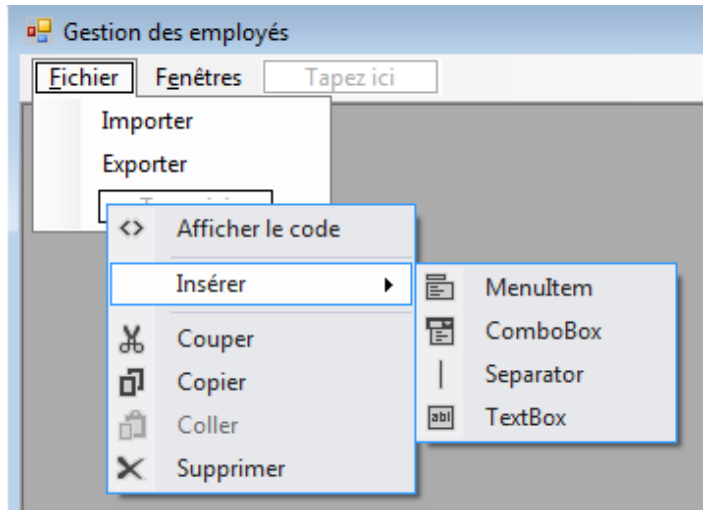


  - Définir la barre de menus



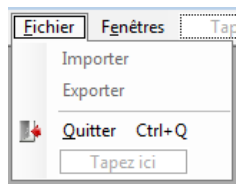
Menu

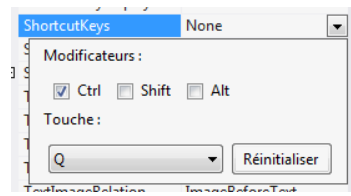
- Utiliser le concepteur de menu

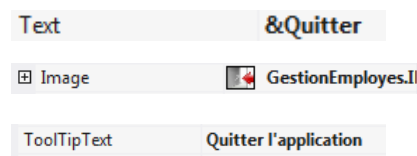


Menu

- Personnaliser l'option de menu







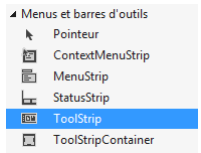
Menu

- Ouvrir une fenêtre fille

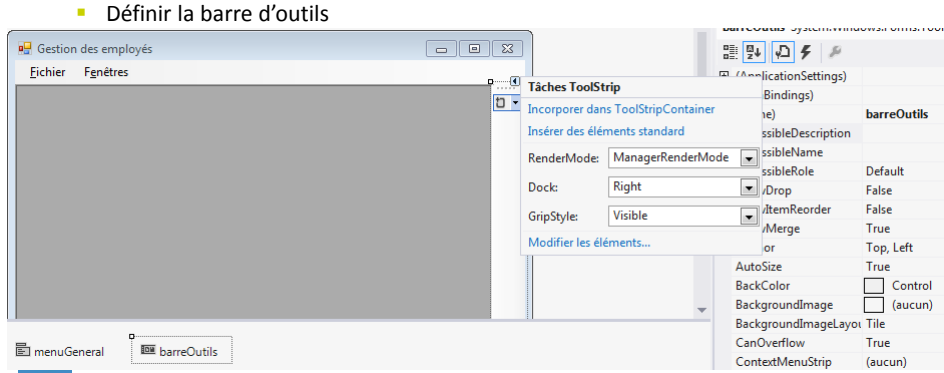
```
frm = new FrmServices();
frm.MdiParent = this;
frm.Show();
```


Barre d'outils

- Le contrôle **ToolStrip** RI 323

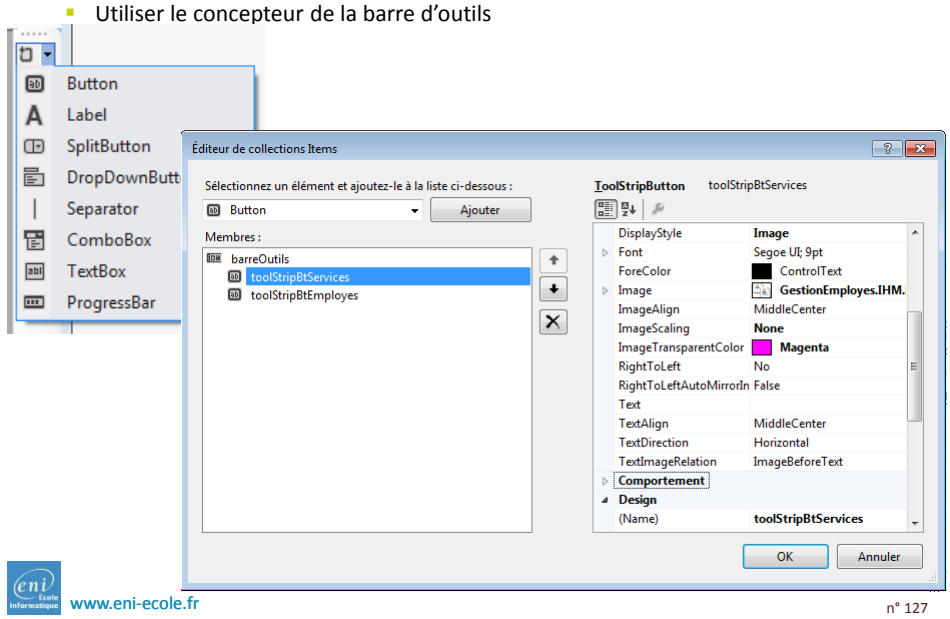


  - Définir la barre d'outils



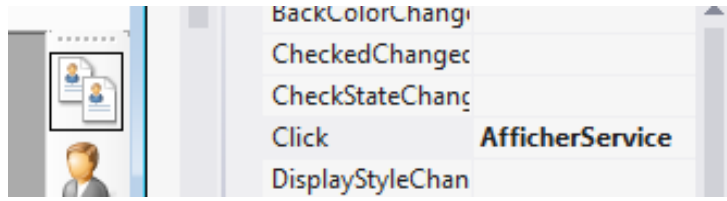
Barre d'outils

- Utiliser le concepteur de la barre d'outils



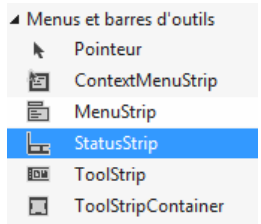
Barre d'outils

- Associer les boutons de la barre d'outils aux écouteurs d'évènement.

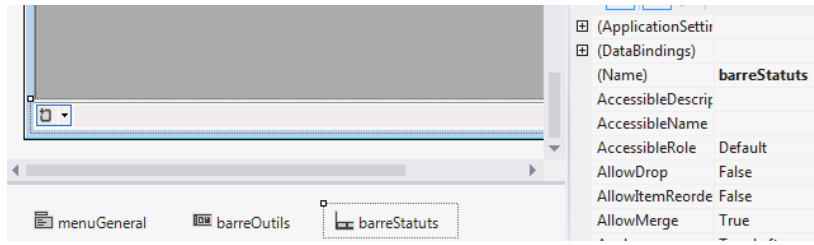


Barre de statuts

- Le contrôle **StatusStrip** RI 307

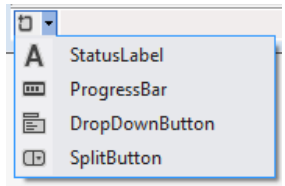


  - Définir la barre de statuts



Barre de statuts

  - Utiliser le concepteur

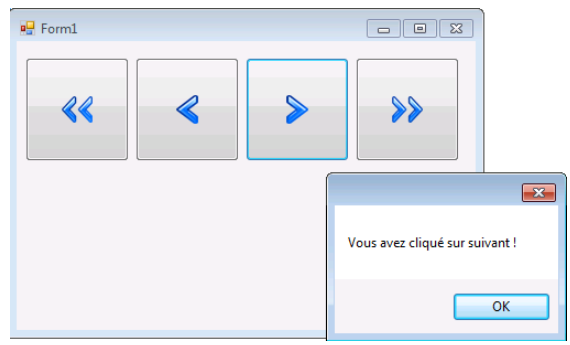


Créer un contrôle utilisateur

- L'objectif d'un contrôle utilisateur est de regrouper de manière logique des contrôles afin d'obtenir une entité graphique réutilisable.
  - Projet de type **Bibliothèque de contrôles Windows Forms**.
  - Classe de base **UserControl**.
- Ce composant pourra être associé à :
  - une interface graphique,
  - des propriétés spécifiques,
  - des évènements spécifiques,
  - la barre d'outils afin d'être utilisé dans d'autre projet.
- **C# - Fiche technique - Créer un contrôle utilisateur**

Exercice dirigé

- A partir de la fiche technique qui vous est proposée, réaliser le contrôle utilisateur représentant une barre de navigation.



Exercice

- Créez la fenêtre **FrmEmployes** et définissez son interface graphique.

Maintenance évolutive

- Modifiez le projet afin de gérer la photo de l'employé

Fin de la séquence de formation