

# Créer son propre moteur de Binding.

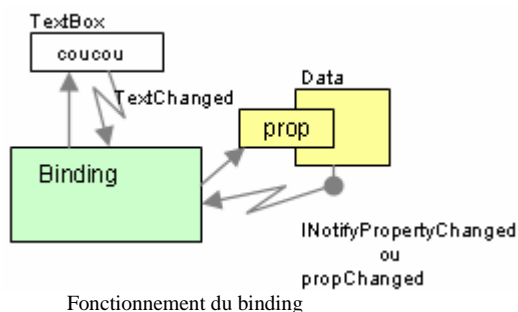
## Introduction

Aujourd'hui dans nos applications Windows Forms, WPF, la connexion de la couche métier à notre interface utilisateur peut se faire en utilisant un mécanisme : le « binding ». Ce mécanisme permet de mettre à jour un objet métier lorsqu'un utilisateur effectue un changement dans l'interface utilisateur. Malheureusement le binding fournit par le framework .Net ne peut pas être utilisé entre deux objets métiers. Nous allons voir comment on peut réaliser un moteur de binding afin de relier des objets métiers entre eux, avec un minimum de contraintes.

## Rappel

En Windows Forms, le binding fonctionne entre un control visuel et n'importe quelle instance d'objet. En WPF le binding fonctionne entre une DependencyProperty et n'importe quelle instance d'objet.

Pour que le binding soit bidirectionnel il faut que l'objet métier implémente INotifyPropertyChanged (ou un événement « nom de la propriété »Changed).



En simplifiant au maximum le binding c'est :

Quand l'utilisateur change le texte de la « TextBox » le code suivant est exécuté :

```
Data.Prop1 = TextBox.Text ;
```

Et dans l'autre sens lorsque ce code est exécuté :

```
Data.Prop1 = « tagada » ;
```

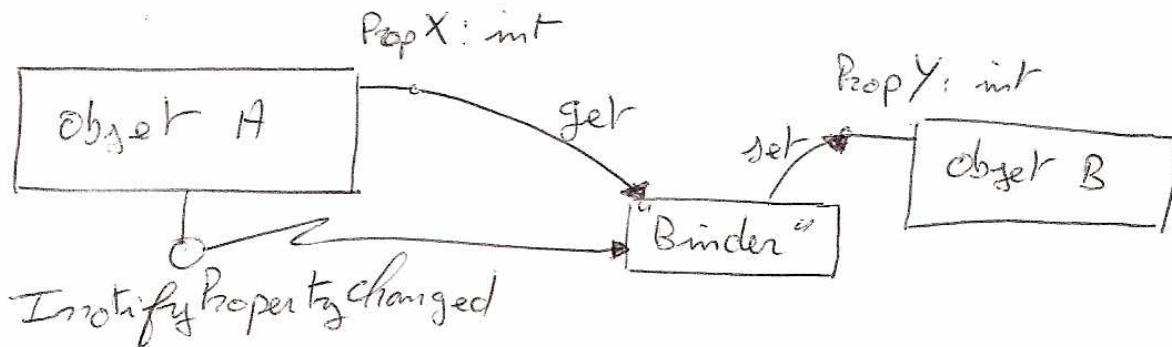
Le moteur du binding execute alors

```
TextBox.Text = Data.Prop1 ;
```

En résumé « Prop1 » est mise à jour quand le texte du control change, et le texte du contrôle est mis à jour lorsque que la propriété est modifiée.

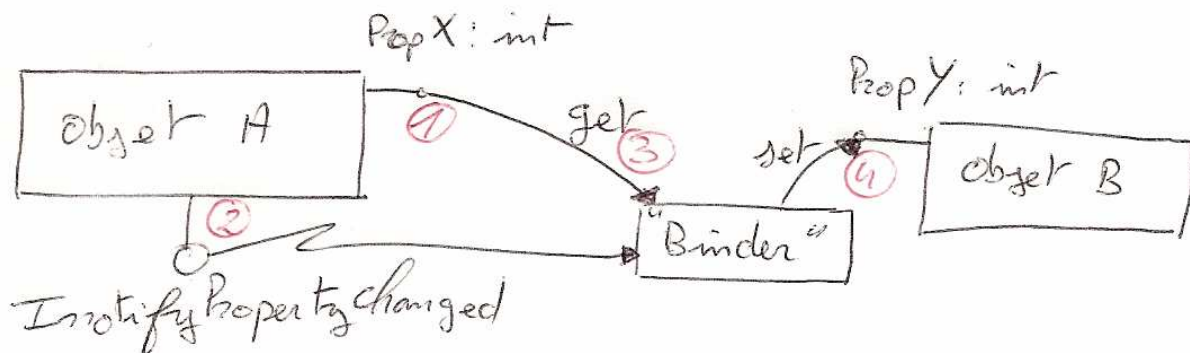
## Les bases du moteur de binding

Le but de créer son moteur est de pouvoir « binder » deux propriétés de deux objets quelconques. L'architecture technique ressemble à :



Evidemment il est inutile de réinventer la roue, utilisons les mécanismes proposés par le Framework .Net. Notre moteur sera donc basé sur « INotifyPropertyChanged ».

Comment fonctionne-t-il ?



- 1) Modification de la propriété source (i.e : A.PropX = 123 ;)
- 2) L'événement « PropertyChanged » est soulevé
- 3) Le « Binder » ayant été notifié via « INotifyPropertyChanged », effectue un « Get » de la propriété PropX sur l'objet A
- 4) Le « Binder » effectue un « Set » sur la propriété PropY de l'objet B.

Une version très simpliste de notre moteur pourrait ressembler à :

```
void AddBinding(INotifyPropertyChanged source, string sourcePropName,
               object destination, string destPropName)
{
   PropertyDescriptor pdSource = TypeDescriptor.GetProperties(source)[sourcePropName];
    PropertyDescriptor pdDest = TypeDescriptor.GetProperties(destination)[destPropName];
    source.PropertyChanged += delegate(object sender, PropertyChangedEventArgs e)
    {
        if (e.PropertyName == sourcePropName)
            pdDest.SetValue(destination, pdSource.GetValue(source));
    };
}
```

Cette version a l'avantage d'être très simple, mais possède en revanche plusieurs défauts majeurs !

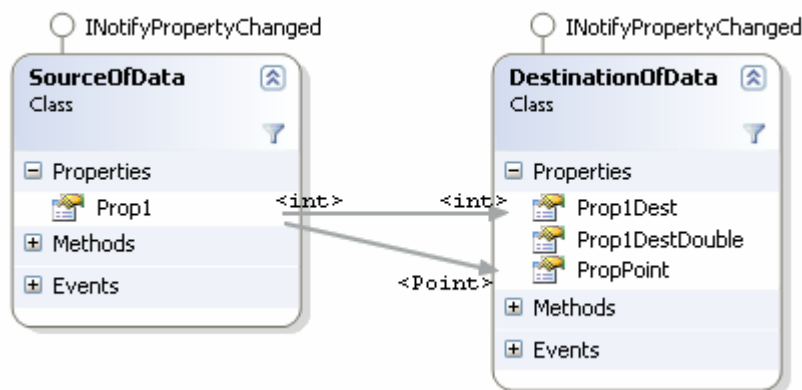
- 1) En effet l'utilisation des déléguées anonymes, ne nous permet de nous désabonner...
- 2) Les objets source et destination sont maintenus en « vies ».
- 3) Chaque « AddBinding », augmente le nombre d'abonnements relatif à « INotifyPropertyChanged ».
- 4) Elle utilise la réflexion, comme le binding standard .Net, ce qui en termes de performances réduit son champ d'action. Il ne serait pas très judicieux d'employer ce mécanisme pour du temps réel.
- 5) Comme pour le binding du Framework .Net (WinForms, WPF) le type utilisé est « object », ce qui provoque du boxing-unboxing lors du binding de types primitifs, et donc nuit à la performance.

Cette version nous servira de base pour la suite de notre moteur, le principe restera le même, mais nous allons corriger tous ces défauts.

## Les génériques au service du binding

Les génériques vont nous aider à résoudre la problématique du boxing-unboxing. Notre moteur devra relier deux propriétés en respectant leur « Type ».

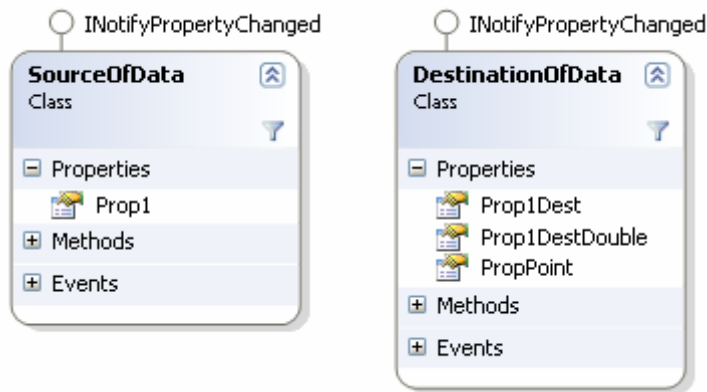
Ce postulat nous amène à éliminer l'utilisation de la réflexion pour une raison simple, elle utilise uniquement le type « object ». Ce choix va aussi dans le sens de la performance.



Exemple je veux « binder » deux propriétés (**int Prop1 -> int Prop1Dest**) et/ou (**int Prop1 -> Point PropPoint**). Un problème persiste, comment obtenir/définir une propriété sans passer par « object », tout en effectuant le binding au runtime ?

## Le LCG au service du binding

Une réponse est effectivement d'employer le LCG (Lightweight Code Generation). Nous utiliserons les méthodes dynamiques dans le but de générer du code fortement typé, ce qui aura pour effet d'éviter l'utilisation de la réflexion.



Voici le code correspondant à la classe SourceOfData :

```
class SourceOfData : INotifyPropertyChanged
{
    private int _prop1;

    public int Prop1
    {
        get { return _prop1; }
        set
        {
            if (_prop1 != value)
            {
                _prop1 = value;
                DoPropertyChanged("Prop1");
            }
        }
    }

    #region INotifyPropertyChanged Members
    protected void DoPropertyChanged(string propName)
    {
        if (PropertyChanged != null)
            PropertyChanged(this, new PropertyChangedEventArgs(propName));
    }

    public event PropertyChangedEventHandler PropertyChanged;
    #endregion
}
```

Il faut donc générer une méthode dynamique qui effectue un « get » sur un entier.

```
delegate int GetIntHandlerDelegate(object source);

static GetIntHandlerDelegate CreateGetHandler(PropertyInfo propertyInfo)
```

```
{
    //Récupération du "getter"
    MethodInfo getMethod = propertyInfo.GetGetMethod(true);
    //création de la méthode dynamique
    DynamicMethod dynamicGet = new DynamicMethod("DynamicGet" + propertyInfo.Name,
                                                typeof(int),
                                                new Type[] { typeof(object) },
                                                propertyInfo.DeclaringType,
true);
    ILGenerator getGenerator = dynamicGet.GetILGenerator();

    getGenerator.Emit(OpCodes.Ldarg_0);
    getGenerator.Emit(OpCodes.Call, getMethod);
    getGenerator.Emit(OpCodes.Ret);

    //création d'une déléguée pour appeler le code généré
    return
(GetIntHandlerDelegate)dynamicGet.CreateDelegate(typeof(GetIntHandlerDelegate));
}
```

La technique consiste à récupérer le « getter » de la propriété du type concerné (*SourceOfData*), et de créer une « déléguée » autour.

La déléguée ainsi créée est liée au type et non à l'instance de *SourceOfData*, nous utiliserons donc la même déléguée pour toutes les instances de *SourceOfData*.

La méthode dynamique générée ressemble à ceci :

```
static int DynamicGetProp1(object source)
{
    return source.Prop1;
}
```

Evidemment on utilise la même technique pour modifier la propriété destination (*DestinationOfData*).

```
static SetHandlerDelegate<TValue> CreateSetHandler<TValue>(PropertyInfo propertyInfo)
{
    MethodInfo setMethod = propertyInfo.GetSetMethod(true);
    DynamicMethod dynamicSet = new DynamicMethod("DynamicSet" +
propertyInfo.Name,
                                                typeof(void),
new Type[] { typeof(object),
typeof(TValue) },
                                                propertyInfo.DeclaringType, true);
    ILGenerator setGenerator = dynamicSet.GetILGenerator();

    setGenerator.Emit(OpCodes.Ldarg_0);
    setGenerator.Emit(OpCodes.Ldarg_1);
    setGenerator.Emit(OpCodes.Call, setMethod);
    setGenerator.Emit(OpCodes.Ret);

    Type tDelegate = typeof(SetHandlerDelegate<TValue>);
    return (SetHandlerDelegate<TValue>)dynamicSet.CreateDelegate(tDelegate);
}
```

La méthode dynamique générée ressemble à :

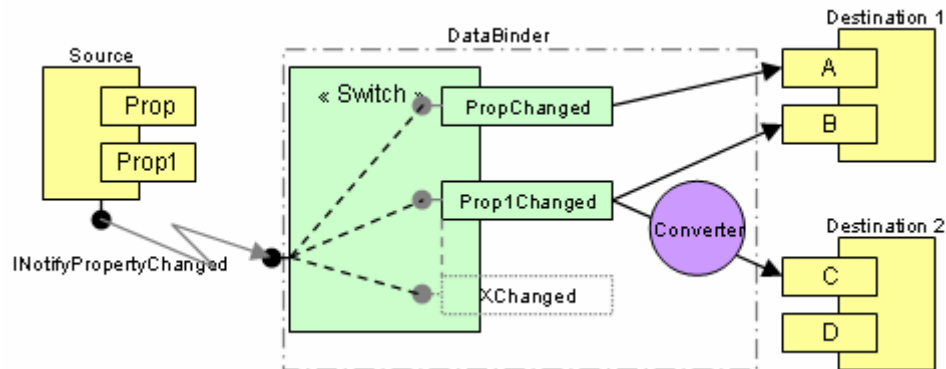
```
static void DynamicSetProp1Dest(object dest, int value)
{
    dest.Prop1Dest = value;
}
```

La génération de méthodes dynamiques va nous permettre d'effectuer un binding compilé et fortement typé !

Nota : Si vous ne connaissiez pas le LCG, il faut savoir qu'il est à la base des langages dynamiques sous .Net (cf. DLR -> Dynamic Language Runtime i.e : IronPython, IronRuby)

## Evolutions

Voici l'architecture globale du moteur de binding



Des évolutions ont été apportées dans la version finale

- 1) Un seul abonnement à `INotifyPropertyChanged` par source, avec une gestion d'un cache des abonnements.
- 2) Possibilité de « binder » avec un convertier (`IBinderConverter`, pas de `TypeConverter` car il n'est pas fortement typé), pour effectuer un binding entre deux types différents.
- 3) Possibilité de retirer un binding.
- 4) Possibilité d'activer/désactiver le binding sans le retirer.
- 5) Gestion de la récursivité.
- 6) Possibilité d'avoir un binding sur une hiérarchie d'objets (voir ci-après).
- 7) Possibilité d'avoir un binding sur un tableau (voir ci-après).

Des évolutions restent à faire

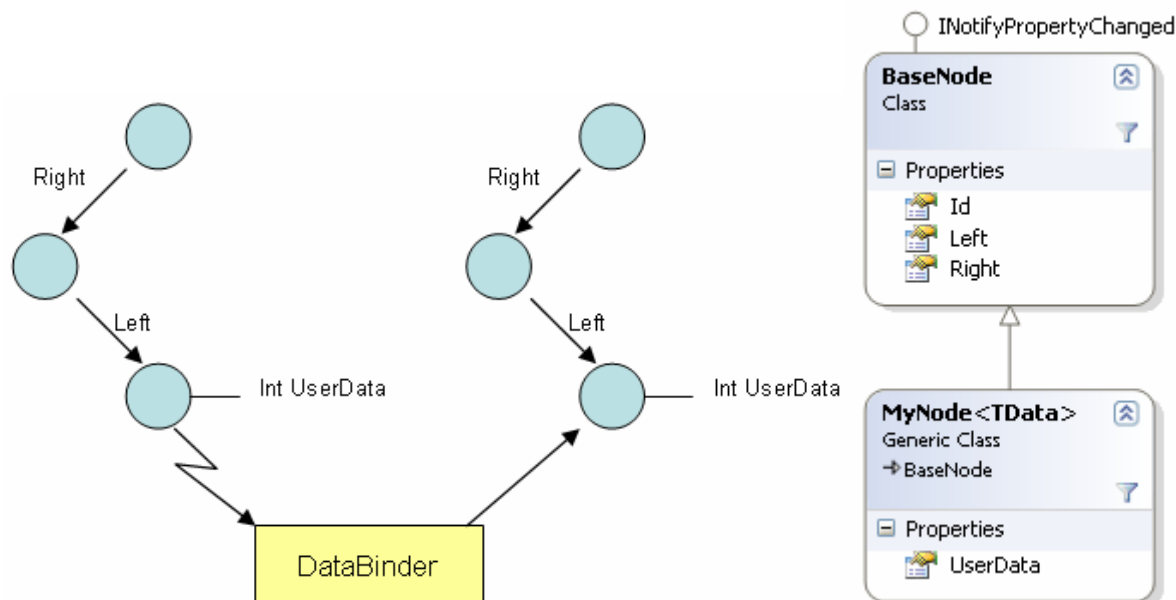
- Il faudrait ajouter le support de « **nom de propriété** » **Changed** en plus de « `INotifyPropertyChanged` », afin de pouvoir utiliser ce moteur pour binder sur des contrôles.
- Il faudrait ajouter le support du binding sur les dictionnaires.

### Binding hiérarchique

Comme je l'ai indiqué ci-dessus, ce moteur est capable de « binder » deux propriétés dans une hiérarchie d'objets. Pour ceux qui connaissent WPF (Windows Presentation Foundation), le principe est identique.

En effet il est possible de « binder » deux objets faisant partie d'une hiérarchie.

Prenons un exemple :

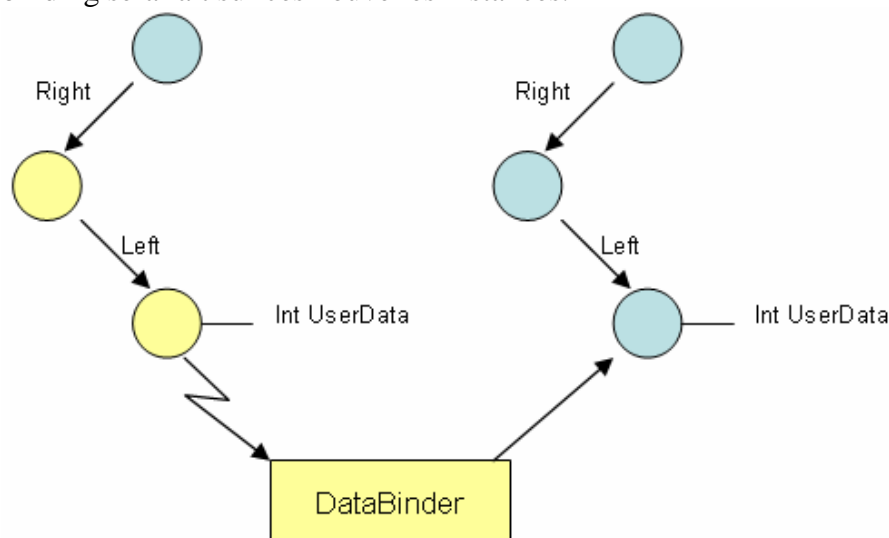


Avec un binding simple, on récupère l'instance de chaque feuille et on « binde » la propriété « UserData » source vers la propriété « UserData » destination. Mais si retire et ajoute des nœuds à notre structure, mon binding simple est devenu obsolète. Souvent dans ce genre de structure de données, seule la racine est connue, il serait donc plus judicieux d'avoir un binding relatif à la racine. Alors comment peut-on effectuer un tel binding ?

```
DataBinder.AddCompiledBinding(source, "Left.Right.UserData", destination, ...);
```

Ici on utilise la notation par «.»(Idem qu'en WPF), pour accéder aux différentes instances des objets sous-jacents, Ainsi "Left.Right.UserData" représente l'instance de la feuille.

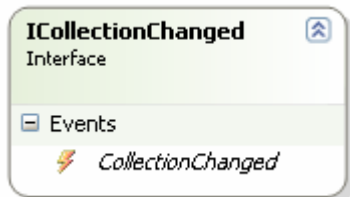
Grace à ce mécanisme, si les instances comprises entre la feuille et la racine « change », le binding sera fait sur ces nouvelles instances.



Enfin pour coller au maximum aux principes de binding proposé par WPF, ce moteur est capable de « binder » aussi des éléments d'un tableau (plus précisément sur un index)

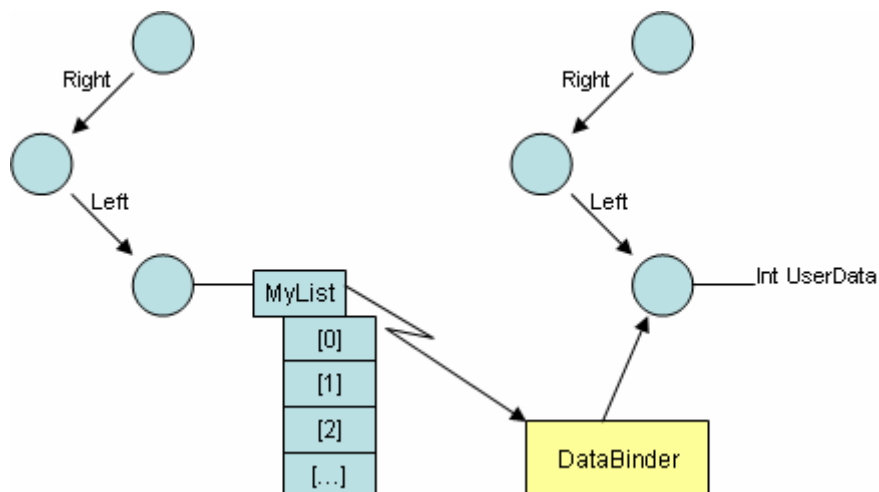
```
DataBinder.AddCompiledBinding(source, "Left.Right.UserData.MyList[0]",  
destination, ...);
```

Evidemment pour que le binding fonctionne correctement le tableau doit être capable de notifier les changements opérés sur celui-ci, ce qui est le cas de la « BindingList ». Pour plus de souplesse ce framework propose une interface :



Le nom de cette interface est proche de celle proposée par le Framework .Net 3.0, c'est volontaire, puisque ce moteur ne s'appuie que sur le Framework 2.0.

Ci-dessous le schéma représentant un binding hiérarchique avec un tableau.



Le moteur est abonné à la liste via « ICollectionChanged », ou via « IBindingList ». Ainsi dès que la liste notifiera du changement sur son index 0, la destination sera mise à jour.



## Conclusion

Le binding n'est pas réservé qu'aux interfaces utilisateur. Nous avons vu qu'il est possible de faire un moteur de binding dans l'esprit de celui fourni par le Framework .Net. L'intérêt ? Réaliser des liens « paramétrable » entre objets.

Ce moteur n'est pas parfait, puisqu'il impose quelques contraintes au niveau du modèle de données :

- 1) les données à surveiller doivent implémenter `INotifyPropertyChanged`. cette contrainte est aussi valable lorsque l'on utilise le binding avec des contrôles windows forms ou WPF
- 2) une contrainte qui n'est pas forcément visible à première vue, les propriétés bindées doivent être réelles. En effet ce moteur n'est pas capable de binder des propriétés générées par un « `CustomTypeDescriptor` ».

Comme tout système, il y a des avantages et inconvénients, et celui-ci n'échappe pas à la règle.

### *Avantages*

- Le binding compilé est ~300 fois plus rapide que le binding par réflexion (c.f. annexe).
- Binding d'objet à objet.
- Pas de boxing-unboxing
- Il ne garde pas de références directes vers les objets source et destination, donc il ne provoque pas de fuite de mémoire.
- Possibilité d'activer/désactiver sans « unbind »
- Possibilité de binder une hiérarchie d'objets
- Possibilité de binder sur un index de tableau

### *Inconvénients*

- les contraintes décrites ci-dessus peuvent être vues comme des inconvénients dans certains cas.
- Il est impossible de binder sur des propriétés « virtuelles » (cf. `ICustomTypeDescriptor`)

Vous trouverez le code complet de ce moteur sur [codeplex.com](http://www.codeplex.com/GeniusBinder)  
<http://www.codeplex.com/GeniusBinder>

## Annexe

### ***Comparaison entre le binding compilé et le binding par reflection.***

J'ai effectué la comparaison entre le binding compilé et le binding par reflexion au niveau du moteur lui-même, c'est-à-dire lorsque celui-ci effectue un « get » sur la source et un « set » sur la destination.

```
private static long TestCompiled()
{
    DataTestForPerf src = new DataTestForPerf();
    DataTestForPerf dst = new DataTestForPerf();

    PropertyInfo piSource = src.GetType().GetProperty("Prop1");
    PropertyInfo piDest = dst.GetType().GetProperty("Prop1");

    GetHandlerDelegate<int> getSrc =
    GetSetUtils.CreateGetHandler<int>(piSource);
    SetHandlerDelegate<int> setDst =
    GetSetUtils.CreateSetHandler<int>(piDest);

    src.PropertyChanged += delegate
    {
        setDst(dst, getSrc(src));
    };

    Stopwatch watch = new Stopwatch();
    watch.Start();
    for (int i = 1; i <= NBTURNS; i++)
    {
        src.Prop1 = i;
    }
    watch.Stop();
    return watch.ElapsedMilliseconds;
}

private static long TestReflection()
{
    DataTestForPerf src = new DataTestForPerf();
    DataTestForPerf dst = new DataTestForPerf();

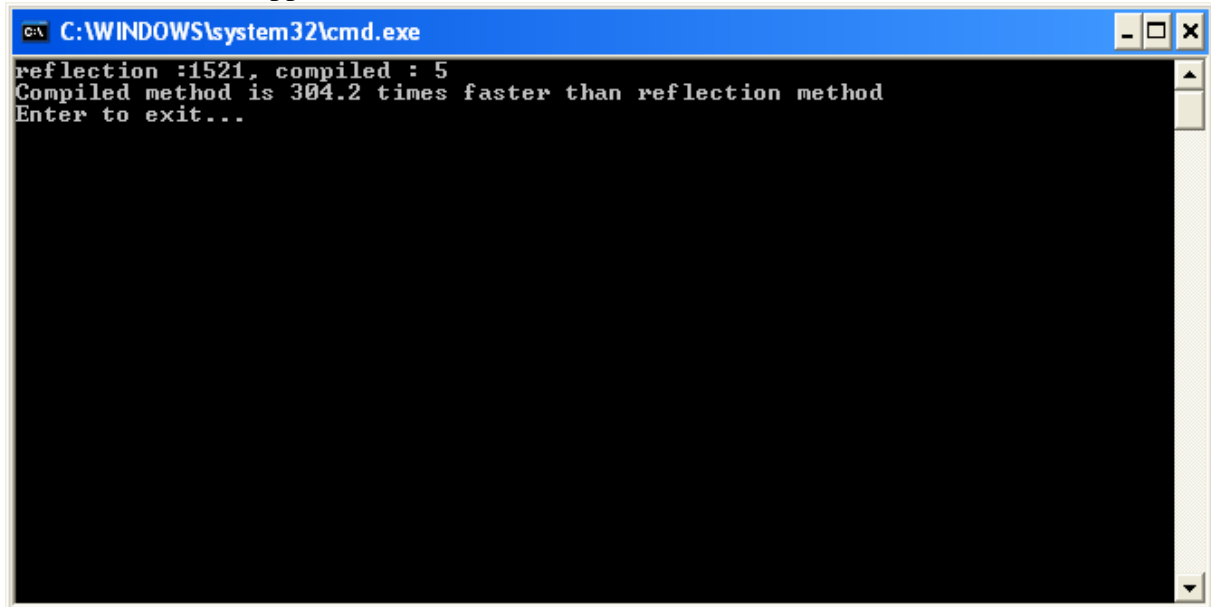
    PropertyDescriptor pdSource =
    TypeDescriptor.GetProperties(src)["Prop1"];
    PropertyDescriptor pdDest =
    TypeDescriptor.GetProperties(dst)["Prop1"];
    src.PropertyChanged += delegate
    {
        pdDest.SetValue(dst, pdSource.GetValue(src));
    };

    Stopwatch watch = new Stopwatch();
    watch.Start();
    for (int i = 1; i <= NBTURNS; i++)
    {
        src.Prop1 = i;
    }
    watch.Stop();
    return watch.ElapsedMilliseconds;
}
```

}

On remarque en **jaune** le code exécuté à chaque modification de la propriété source, et en **violet** le code du test (identique dans les deux cas).

Le résultat est sans appel .....



A screenshot of a Windows command prompt window titled "C:\WINDOWS\system32\cmd.exe". The window has a blue title bar and standard Windows window controls. The text inside the window reads: "reflection :1521, compiled : 5", "Compiled method is 304.2 times faster than reflection method", and "Enter to exit...".

Oui vous lisez bien 300 fois plus rapide !!!

Le code source complet du test est aussi disponible  
sur <http://www.codeplex.com/GeniusBinder>.

Configuration de la machine de test :

