intech
ECOLE SUPERIEURE
D'INGENIERIE INFORMATIQUE

Exercice : Des Recettes Concrètes

invenietis

1 Enoncé

Cet exercice est fondé sur ITI.SimpleRecipes et lui ressemble beaucoup fonctionnellement. Ce qui diffère est que le modèle n'est constitué que de contrats (interfaces) : l'implémentation du modèle est totalement masquée, seules les interfaces sont visibles.

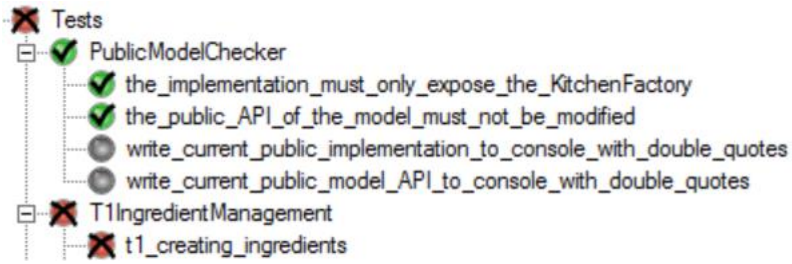
Le but de cet exercice est de pratiquer l'implémentation d'interface, d'acquérir quelques réflexes en termes d'encapsulation et, d'introduire une notion importante : la covariance et de manipuler un peu les chaînes de caractères avec deux algorithmes d'importation.

Vous disposez pour cela d'une solution comprenant 3 projets :

- ITI.SimpleRecipesV2.Tests, contient les tests unitaires à valider
- ITI.SimpleRecipesV2.Model, contient les abstractions : des interfaces, des classes abstraites, des enums (et éventuellement quelques classes concrètes utilitaires, comme des EventArgs ou desq exceptions).
- ITI.SimpleRecipesV2.Impl, contient l'implémentation du modèle. Cet assembly expose un unique point d'entrée « contractuel » qui est une classe statique qui permet la création et le chargement d'un IKitchenContext.
  Cette assembly ne contient initialement que cette classe. Rien d'autre que cette classe (avec ses 2 méthodes) ne doit être public.

Pour lancer les tests unitaires il vous suffit de configurer le projet ITI. SimpleRecipesV2.Tests en tant que projet de démarrage puis d'exécuter la solution.

Comme vous pouvez le constater, pour le moment, tous les tests sont rouges, à l'exception de 2 d'entre eux :

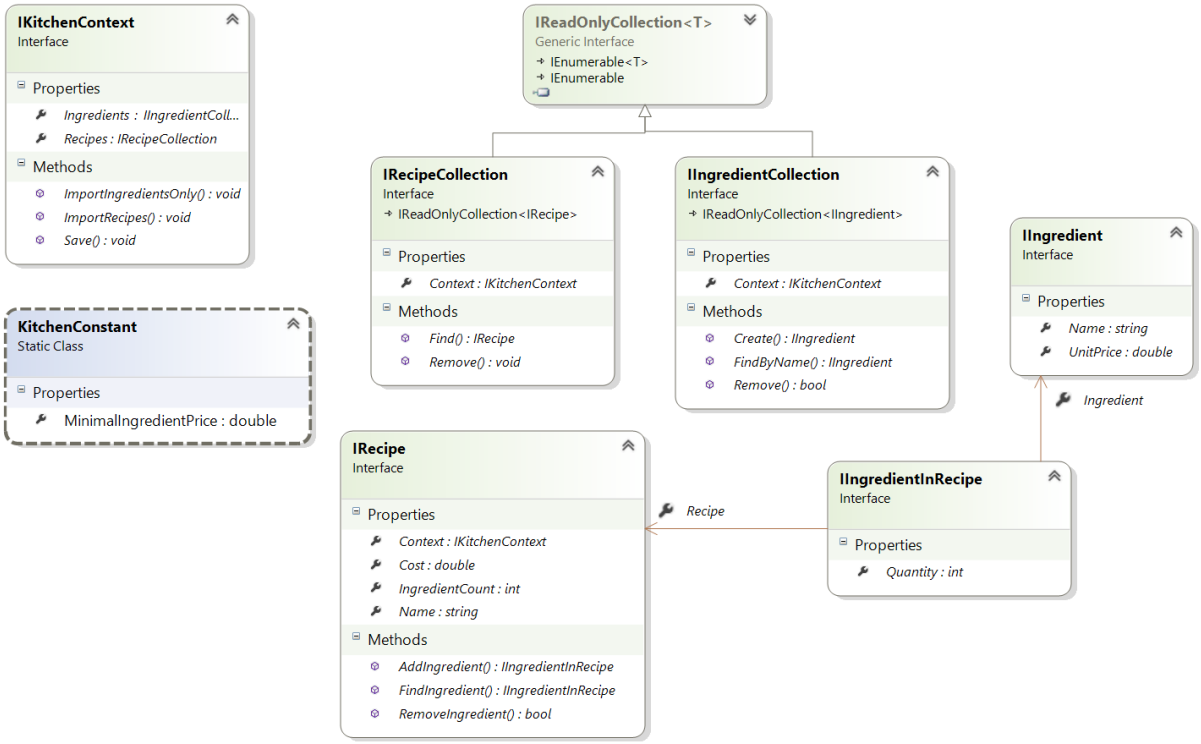


Ces deux tests analysent les objets implémentés dans ITI.SimpleRecipesV2.Model.dll et ITI.SimpleRecipesV2.Impl.dll et en comparent les API publics (les classes et leurs membres) à leur version initiale respective. Si d'aventure vous rendez public et/ou modifiez l'API public, ces tests seront rouges.

Pour les autres tests, à vous de faire en sorte qu'ils passent en vert. Pour cela, vous avez le droit de faire ce que bon vous semble dans le projet ITI. SimpleRecipesV2.Impl. (Vous ne devez évidemment pas modifier le projet ITI.SimpleRecipesV2.Tests.)

Les tests unitaires sont là pour spécifier de façon détaillée les fonctionnalités attendues. Ce qui est attendu reprend SimpleRecipes en y ajoutant :

- Les deux collections principales (IRecipeCollection et IIngredientCollection) sont des IReadOnlyCollection<T>, on peut donc manipuler ces collections de façon standard (foreach, LINQ, etc.)
- Une sérialisation/dé-serialisation d'un IKitchenContext. Cette sérialisation peut être de n'importe quel type : XML, binaire, JSON, texte, etc. tant que cela fonctionne.
- Deux imports de données depuis des fichiers textes : les ingrédients seulement (`IKitchenContext.ImportIngredientsOnly`) et les recettes et leurs ingrédients (`IKitchenContext.ImportRecipes`).

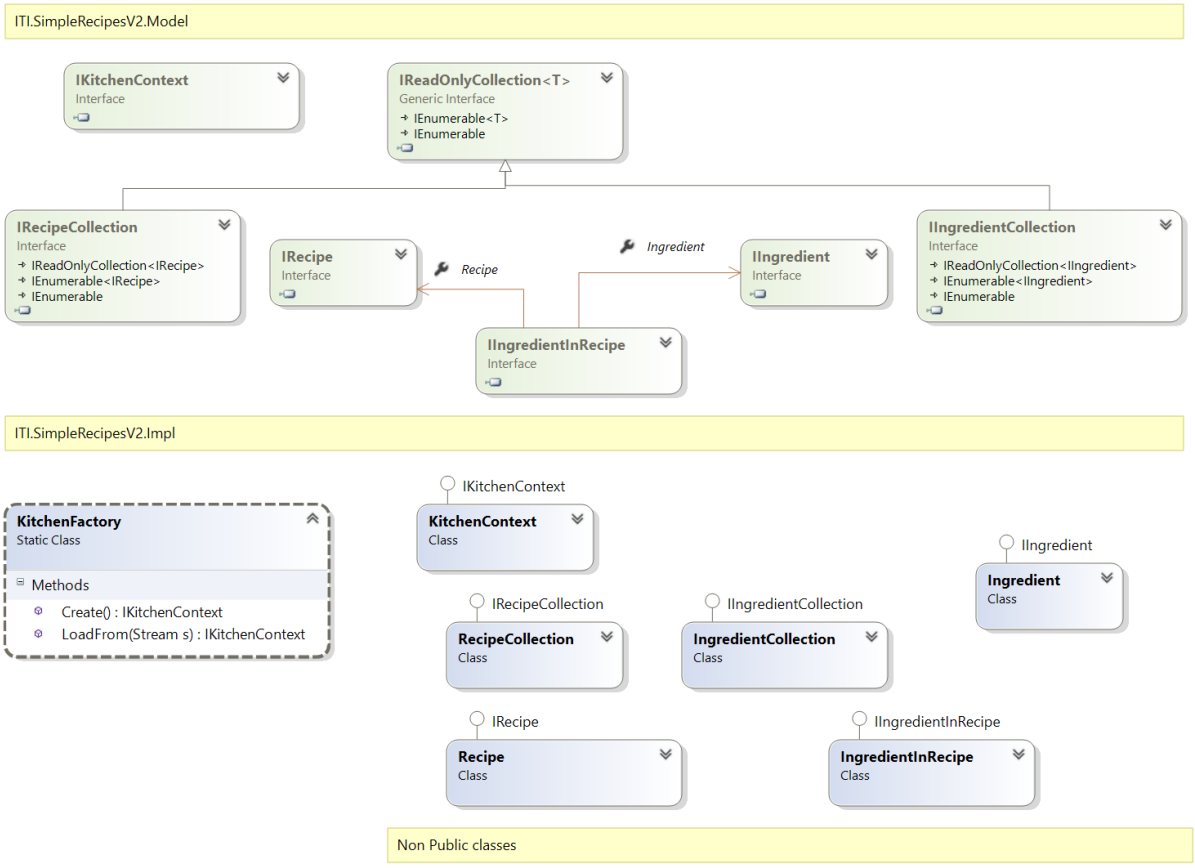


Normalement, vous pouvez commencer : une maîtrise correcte de C# permet de passer à l'implémentation immédiatement, mais il faut connaître l'implémentation explicite d'interface et savoir mettre en œuvre le downcasting à bon escient.

Essayez… ou passez à la page suivante pour plus d'informations sur ces aspects.

2 Abstractions à l'extérieur, Implémentations à l'intérieur

Le principe de cet exercice est d'encapsuler totalement l'implémentation. A l'extérieur, le développeur-utilisateur manipule un IRecipe, à l'intérieur vous manipulez un Recipe :



Lorsque votre code « sort » une référence à un objet, il n'y a aucun problème : une fonction telle que celle-ci (qui a accès à une `List<Recipe>` _recipes) compile et fonctionne parfaitement bien :

```
public IRecipe GetTimeBasedRandomRecipe()
{
    Recipe r = _recipes[ (int)(DateTime.UtcNow.Ticks % _recipes.Count) ];
    return r;
}
```

Simplement parce qu'une instance de `Recipe` EST_UN `IRecipe`. C'est le principe de base de la spécialisation.

A l'intérieur, vous devez manipuler directement les classes d'implémentations.

Malheureusement, il arrive d'avoir besoin de « rentrer » une abstraction depuis l'extérieur vers l'intérieur, ce qui est le cas de `IRecipe`.`AddIngredient` :

```
public IIngredientInRecipe AddIngredient( IIngredient ii, int quantity = 1 )
{
      //...
}
```

Pour retrouver l'implémentation, c'est simple : il suffit de « downcaster » :

```
Ingredient i = (Ingredient)ii;
```

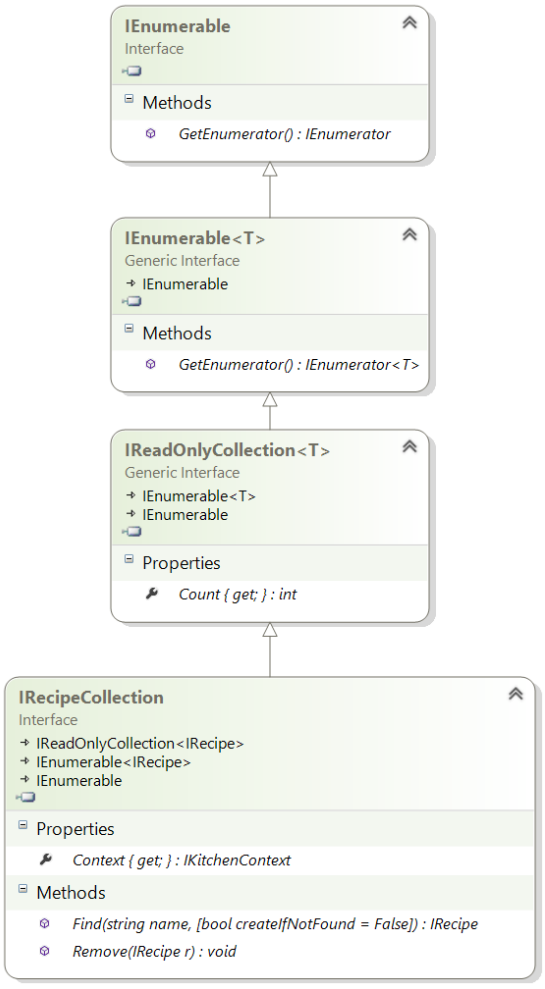
Et ensuite de travailler avec `i` ce qui permet d'accéder à toutes ses propriétés et méthodes. Cela dit, si vous travailler dans un monde « ouvert », rien n'empêche un autre développeur, une autre équipe de développer une autre implémentation du modèle de cuisine.
Si un tel `IIngrédient`, implémenté dans un autre assembly, est passé ici, le cast ci-dessus provoquera un horrible `InvalidCastException`. Afin d'éviter cela, il suffit d'utiliser l'opérateur `as` :

```
Ingredient i = ii as Ingredient;
```

Avec cet opérateur, `i` sera (gentiment) `null` si `ii` n'est effectivement pas un `Ingredient` à nous.

3 Implémentation explicite d'interface

L'implémentation explicite d'interface est un moyen de résoudre les conflits de nommage. Les collections doivent implémenter IReadOnlyCollection<T>, donc IEnumerable<T>, donc l'interface non-générique IEnumerable d'origine du .Net framework 1.0.



Concrètement, ce n'est vraiment pas grand-chose : la propriété Count (en get) et les 2 méthodes GetEnumerator(), une qui doit retourner le non-générique `IEnumerator` et un deuxième qui lui doit retourner un `IEnumerable<IRecipe>`.

Problème : ces deux méthodes ont le même nom ET les mêmes paramètres. On ne peut faire cela :

```
public IEnumerator GetEnumerator()
{
    return ...;
}

public IEnumerator<IRecipe> GetEnumerator()
{
    return ...;
}
```

Car comment le compilateur pourrait-il savoir quelle méthode appeler (polymorphisme *ad hoc*) ?

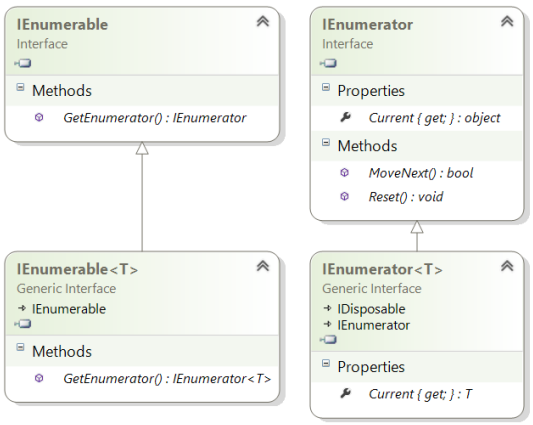
Le C# permet de résoudre ces conflits de nommage de façon simple : les méthodes des interfaces (que notre objet est obligé d'implémenter - c'est le principe fondamental des interfaces) peuvent être implémentées explicitement pour chaque interface. Il suffit pour cela de nommer l'interface devant le nom de la méthode :

```
IEnumerator IEnumerable.GetEnumerator()
{
    return ...;
}


IEnumerator<IRecipe> IEnumerable<IRecipe>.GetEnumerator()
{
    return ...;
}
```

Ci-dessus les deux méthodes sont implémentées explicitement, ce n'est pas utile : on laisse généralement visible (publique) la version la « plus précise » ou la « meilleure » :

```
IEnumerator IEnumerable.GetEnumerator()
{
    return ...;
}


public IEnumerator<IRecipe> GetEnumerator()
{
    return ...;
}
```

Et pour finir sur ce GetEnumerator, comme on a compris que IEnumerator<T> étend l'interface non-générique IEnumerator :



On peut donc écrire :

```
IEnumerator IEnumerable.GetEnumerator()
{
    return GetEnumerator();
}

public IEnumerator<IRecipe> GetEnumerator()
{
    return ...;
}
```

Ce qui permet de rester très DRY ☺.

Note : Encore plus joli avec les lambda methods du C# 6 (VS2015) :

```
IEnumerator IEnumerable.GetEnumerator() => GetEnumerator();
```

L'implémentation explicite a d'autres usages que celui de résoudre les conflits de noms. Il permet souvent de masquer de la « tuyauterie d'implémentation » aux yeux du développeur-utilisateur d'une API, ou encore, dans notre cas précis, de garder un code propre, dans lequel n'apparait que le strict nécessaire nombre de cast… J'espère qu'en faisant l'exercice vous inventerez vous-même cette jolie astuce.