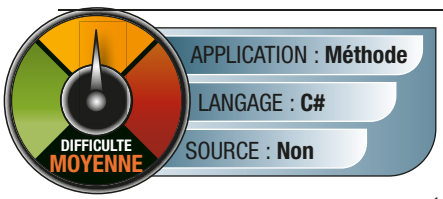
Design By Contract : Code Contracts avec C# 4.0

Dans l'article S.O.L.I.D paru il y a quelques mois, nous vous avons présenté les grands principes qui ont pour objectif d'améliorer la qualité des programmes. Cet article approfondit le principe de Ségrégation de Liskov (LSP) et présente comment Microsoft, par le biais de l'API Code Contracts, permet de l'implémenter de manière efficace et élégante.



Bertrand Meyer, en 1985, est le premier à introduire la technique de programmation par contrat (Design By Contract) permettant de répondre techniquement au Principe de Liskov. Cette technique propose de vérifier que certaines conditions formant un contrat sont vraies à un moment donné d'un programme. Pour cela, trois types de conditions sont définies :

- Pré-condition : condition devant être vraie en amont du traitement
- Post-condition : condition devant être vraie en aval du traitement
- Invariant : condition devant toujours être vraie

1 CODE CONTRACTS

En tant que développeur, il est recommandé de valider tout entrant externe à une méthode. Nombreux sont les développeurs qui passent du temps à implémenter leur propre logique de validation qui vérifie une certaine condition et lève une exception si elle n'est pas respectée. Cette approche peut devenir rapidement illisible et lourde à maintenir. Voici un exemple de validation sans Code Contracts :

```
public double ComputeInterest(double amount, double rate)
{
  double result = 0;

  //Pre-Condition
  if (rate < 1)
    throw new ArgumentOutOfRangeException(
      «Interest rate must be less than 1.»);

  result = amount + amount * rate;

  //Post-Condition
  if (result < amount)
    throw new ArgumentOutOfRangeException(
      «Amount after interest must be superior to initial amount.»);

  return result;
}
```

Même si le code peut être optimisé en encapsulant la logique de validation dans une seule classe, il reste l'exemple de ce qu'il ne faut plus faire car chaque méthode a besoin de sa propre logique de validation. Afin de répondre à cette problématique, Microsoft propose une implémentation du principe « Design by Contract » à travers une API très complète et simple à utiliser : Code Contracts. Tout ce qui est nécessaire à l'utilisation de cette API est déjà présent dans le .NET Framework 4.0 (dans la BCL). Il est aussi possible d'utiliser Code Contracts avec le .NET Framework 3.5, ***en téléchargeant les binaires*** sur le site de Microsoft http://research.microsoft.com/en-us/projects/contracts/. Les différents types de contrats disponibles ainsi que la manière de les utiliser vont être présentés par le biais d'un exemple simple. Un point à remarquer : Pour pouvoir utiliser Code Contracts, l'activation se trouve dans les paramètres de projet dans l'onglet Code Contracts [Fig.1].

2 API CODE CONTRACTS

Voici comment implémenter les différents types de contrats disponibles dans l'API Code Contracts.

Les Pré-conditions

Ce sont les conditions qui doivent être vraies en début de traitement. En général, il faut s'assurer que certains paramètres ne sont pas nuls, ou encore qu'ils ont des valeurs valides. De cette manière, on est capable de savoir que le module est appelé dans des conditions normales de fonctionnement. Voici un exemple de pré-conditions :

```
//Pre-Condition
Contract.Requires(rate > 0, «Rate must be higher than 0.»);
Contract.Requires(amount > 0, «Amount must be higher than 0.»);
Contract.Requires(rate < 1, «Rate must be less than 1.»);
```

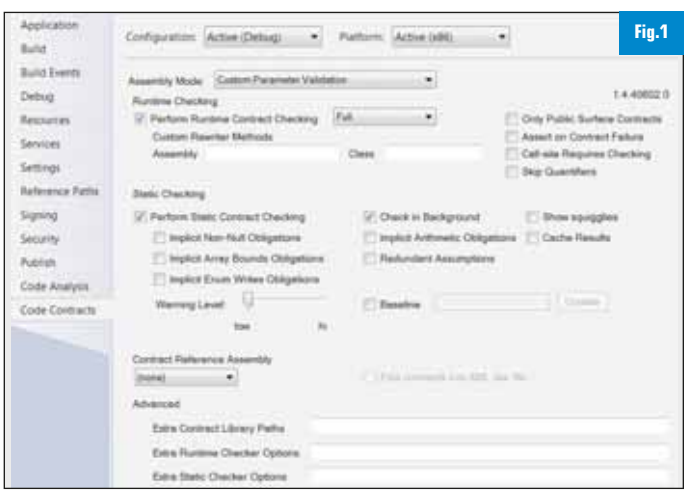


Fig.1

La méthode Requires(…) de la classe statique Contract est utilisée. Les conditions sont positionnées en début de méthode et permettent de spécifier le message d'erreur associé à chaque non-respect de conditions. A l'exécution de la méthode avec des paramètres d'entrée qui ne respectent pas les pré-conditions, une exception est levée au runtime [Fig.2].

Les Post-conditions

Les post-conditions sont les conditions qui doivent être vraies à la fin d'un traitement. Les post-conditions sont utilisées pour valider que le résultat renvoyé est fonctionnellement valide : on va par exemple vérifier que le total d'une facture est supérieur à zéro, que la liste des articles commandés contient bien plusieurs articles…
Pour cela, la classe statique Contract est à nouveau utilisée, cette fois-ci, c'est la méthode Ensure(…) qui est appelée. Comme pour les pré-conditions, les post-conditions sont généralement positionnées en début de méthode. Voici un exemple de post-condition :

```
//Post-Condition
Contract.Ensures(Contract.Result<double>() > amount,
  «Amount after interest must be superior to initial amount.»);
```

A l'exécution de la méthode avec un résultat qui ne respecte pas les post-conditions, une exception est levée au runtime [Fig.3].

Les Invariants

Les invariants sont des conditions qui doivent toujours être respectées. Au lieu d'insérer la validation de ces conditions de multiples fois dans le code, Code Contracts permet de les centraliser. Pour cela, il faut regrouper toutes les conditions invariantes dans une même méthode identifiée avec l'attribut [ContractInvariantMethod]. L'API injecte du code qui permet l'exécution de la validation à chaque appel d'une méthode ou quand une propriété de la classe est modifiée. Attention, car cela peut avoir des impacts sur la performance. Prenons l'exemple suivant : un contrat d'invariance est défini sur une classe, la méthode ContractInvariants(…) contient des conditions sur la propriété ContractID et sur le champ privé _minimum.

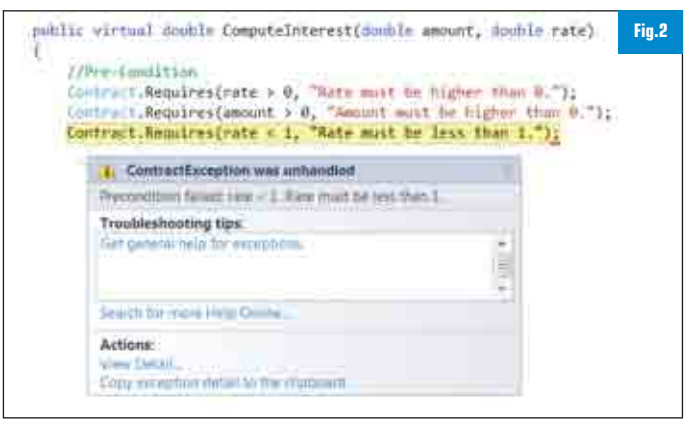


Fig.2

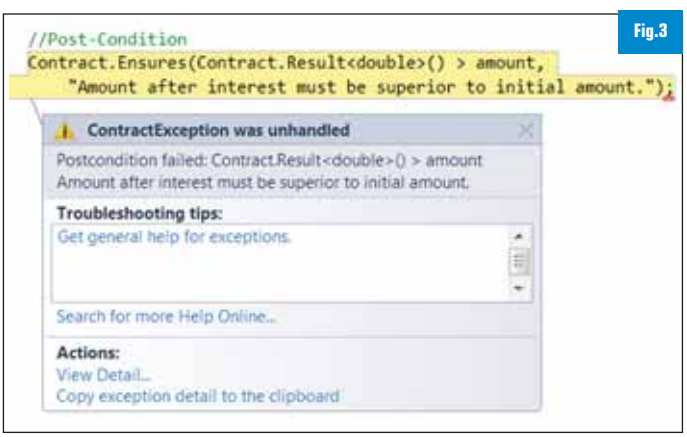


Fig.3

```
private double _minimum = 1;
public string ContractID { get; private set; }

public CEL(string contractID)

{
 ContractID = contractID;
}

[ContractInvariantMethod]
private void ContractInvariants()
{
 Contract.Invariant(_minimum >= 1);
 Contract.Invariant(ContractID.Length == 4);
}
public virtual double ComputeInterest(double amount, double rate)
{
 //Post-Condition
 Contract.Ensures(Contract.Result<double>() > amount,
   «Amount after interest must be superior to initial amount.»);

 return amount + amount * rate + _minimum;
}
```

Regardons de plus près ce qui a été injecté lors de la compilation avec un outil de décompilation tel que .NET Reflector.

```
public virtual double ComputeInterest(double amount, double rate)
{
 double Contract.Old(amount);
 try
 {
   Contract.Old(amount) = amount;
 }
 catch (Exception exception1)
 {
   if (exception1 == null)
   {
     throw;
   }
 }

 double CS$1$0000 = (amount + (amount * rate)) + this._minimum;
 double Contract.Result = CS$1$0000;
 __ContractsRuntime.Ensures(Contract.Result > Contract.Old(amount),
   «Amount after interest must be superior to initial amount.»,
   «Contract.Result<double>() > amount»);
 this.$InvariantMethod$();
 return Contract.Result;
}
```

L'API injecte un appel à la méthode ContractInvariants(…) à la fin de la méthode ComputeInterest(…). Voici ce qui a été injecté dans le construc-

teur de la classe CEL et dans la propriété ContractID (get et set) :

```
public CEL(string contractID)
{
 bool flag = this.$evaluatingInvariant$;
 this.$evaluatingInvariant$ = true;
 this._minimum = 1.0;
 this.ContractID = contractID;
 this.$evaluatingInvariant$ = flag;
 this.$InvariantMethod$();
}

public string ContractID
{
 [CompilerGenerated]
 get
 {
   string Contract.Result = this.<ContractID>k__BackingField;
   if (!this.$evaluatingInvariant$)
   {
     __ContractsRuntime.Ensures(Contract.Result.Length == 4,
       null, «ContractID.Length == 4»);
   }
   return Contract.Result;
 }
 [CompilerGenerated]
 set
 {
   if (!this.$evaluatingInvariant$)
   {
     __ContractsRuntime.Requires(value.Length == 4,
       null, «ContractID.Length == 4»);
   }
   this.<ContractID>k__BackingField = value;
 }
}
```

L'API injecte seulement la vérification des conditions nécessaires en fonction du contexte.

Le fonctionnement des contrats d'invariance se résume donc ainsi :

- Exécution en fin de méthode, donc pas de vérification en début ou au milieu de la méthode.
- Prise en compte des propriétés en injectant les conditions liées aussi bien dans le set que dans le get.

Comme déjà évoqué, les performances de la classe peuvent être dégradées, il est donc nécessaire de bien évaluer ce qui doit être traité comme invariant afin de trouver un juste milieu entre validation fonctionnelle et performance.

3 CODE CONTRACTS ET LE MODÈLE OBJET

Nous avons parcouru les principaux types de contrats. Maintenant, il nous faut voir comment les intégrer à un modèle objet et les combiner avec les principes de la programmation orientée objet tels que l'héritage, le polymorphisme et l'utilisation des interfaces ?

Héritage & polymorphisme de méthode

Lorsqu'on dérive une classe qui possède des Contrats, la classe dérivée hérite automatiquement des contrats définis dans la classe de base. L'héritage des Contrats dans les sous-types est justement ce qui permet de respecter le principe de Liskov : on maîtrise les différentes modifications faites au type de base en s'assurant que le fonctionnement nominal est modifiable dans la limite des contrats définis. Il est également possible de modifier ou de compléter le contrat en ajoutant des conditions en début de méthode. Dans l'exemple suivant, la classe CEL contient la méthode ComputeInterest(…) avec des contrats.

La classe LivretA hérite des contrats et rajoute un nouveau Contrat avec une nouvelle pré-condition.

```
public class CEL
{
 private double _minimum = 1;
 public string ContractID { get; private set; }

 public CEL(string contractID)
 {
   ContractID = contractID;
 }

 [ContractInvariantMethod]
 private void ContractInvariants()
 {
   Contract.Invariant(_minimum >= 1);
   Contract.Invariant(ContractID.Length == 4);
 }

 public virtual double ComputeInterest(double amount, double rate)
 {
   //Pre-Condition
   Contract.Requires(rate < 1, «Rate must be less than 1.»);

   //Post-Condition
   Contract.Ensures(Contract.Result<double>() > amount,
     «Amount after interest must be superior to initial amount.»);

   return amount + amount * rate + _minimum;
 }

 public virtual double ComputeInterest(double amount, double rate,
 double minimum)
 {
   _minimum = minimum;
   return ComputeInterest(amount, rate);
 }
}

public class LivretA : CEL
{
 public LivretA() : base(«9999») { }

 public override double ComputeInterest(double amount, double rate)
 {
   //Additional Pre-Condition
   Contract.Requires(rate > 0, «The interest rate must be higher
than 0.»);
```

```
    return base.ComputeInterest(amount, rate);
  }
}
```

Contrat par interface

Une autre approche de conception consiste en l'association de contrats à une interface, l'idée étant que chaque classe qui implémente cette interface se voit automatiquement imposer ses contrats.

Cela est particulièrement important quand on utilise la programmation basée sur les interfaces (Interface Based Programming). En effet, des contraintes fonctionnelles sont ajoutées aux interfaces en plus des contraintes purement techniques : on évite ainsi les implémentations techniquement valides mais incohérentes.

L'exemple suivant présente une interface et la classe associée qui porte la définition des contrats.

La classe doit respecter les conditions suivantes :

- La classe doit implémenter l'interface
- La classe doit être abstraite (il n'y a aucun sens à vouloir créer une instance de ce type)
- La classe doit porter l'attribut [ContractClassFor(…)]
- Chaque méthode doit porter l'ensemble des contrats

```
[ContractClass(typeof(DefaultOperationClass))]
public interface IInterest
{
  double ComputeInterest(double amount, double rate);
}

[ContractClassFor(typeof(IInterest))]
internal abstract class DefaultOperationClass : IInterest
{
  public double ComputeInterest(double amount, double rate)
  {
    //Pre-Condition
    Contract.Requires(rate < 1, «Rate must be less than 1.»);

    //Post-Condition
    Contract.Ensures(Contract.Result<double>() > amount,
      «Amount after interest must be superior to initial amount.»);

    //Dummy Value, this method only serves to add additional
  contracts
    return 0;
  }
}
```

4 POUR ALLER PLUS LOIN AVEC CODE CONTRACTS

L'utilisation seule de Code Contracts est puissante, comme vous avez déjà pu le remarquer ; Code Contracts peut également être utilisé avec d'autres outils pour devenir un petit bijou au quotidien.

Pex & Code Contracts

Pex est une API de Microsoft Research proposant de générer automatiquement des tests unitaires (http://research.microsoft.com/en-us/projects/pex/). L'utilisation de Code Contracts permet à Pex de proposer des jeux de tests encore plus pertinents et plus complets. Ceci est effectué par lecture des contrats existants pour aboutir à une couverture maximale du code. La combinaison de ces 2 outils est donc très efficace.

SandCastle & Code Contracts

SandCastle est un outil de génération automatique de documentation de code (http://www.codeplex.com/Sandcastle/). Sandcastle est gratuit et peut être étendu pour fonctionner avec Code Contracts. Les fichiers de documentation XML contiennent ainsi des informations supplémentaires sur les contrats. De plus, des feuilles de style peuvent être utilisées avec Sandcastle pour que les pages de documentation générées incluent des sections de contrat.

L'analyse statique du Code

L'analyse statique permet d'évaluer de manière automatique l'exactitude du code source sans l'exécuter. Il existe plusieurs approches pour la mettre en pratique. La plus simple est l'utilisation du compilateur. Static Checker est une autre approche qui est un exécutable s'intégrant dans le processus de build. Le Checker analyse les faits et montre les éventuelles erreurs ainsi que les violations des contrats. L'analyse statique est donc un bon moyen de réduire les temps de développement.

L'attribut [ContractAbbreviator]

Imaginons que votre code contienne des méthodes dans une classe qui a les mêmes paramètres et/ou les mêmes contraintes qui doivent être vérifiées. Il faudrait normalement copier les contrats dans chacune de ces méthodes. Contract abbreviators permet de définir une seule méthode qui contiendra la validation pour les autres méthodes via l'utilisation de l'attribut [ContractAbbreviator].

C'est un attribut assez pratique qui ne se trouve pas dans les binaires contenant l'API Code Contracts. Il faut référencer la classe contenant la définition directement dans les projets (elle se trouve dans le répertoire « %ProgramFiles%/Microsoft/Contracts/Languages/ContractAbbreviator.cs »). Cet attribut permet donc de regrouper les contrats dans une méthode à part et ainsi de les factoriser à un seul endroit.

CONCLUSION

L'API Code Contracts apparaît donc comme une amélioration majeure du framework .NET. Elle permet de gérer en interne les exceptions liées à la validation du code, de positionner les contrats au niveau d'abstraction souhaité sans nuire à la lisibilité du code. Et bien sûr, vous pouvez personnaliser vos contrats et vous assurez de leur respect avant même l'exécution, grâce à l'analyse statique ! L'utilisation de PEX et de SandCastle va permettre d'augmenter la qualité de vos tests et de votre documentation et cela de manière automatisée. Il ne vous reste plus qu'une chose à faire : l'adopter !

Retrouvez le code des exemples sur http://designbycontract.codeplex.com !

■ Jason **De Oliveira**
Practice Manager & Solutions Architect | MVP C#
.Net Rangers by Sogeti
Blog : http://jasondeoliveira.com

■ Fathi **Bellahcene**
Software Architect
.Net Rangers by Sogeti
Blog : http://blogs.codes-sources.com/fathi