

Comment éviter les duplications de code : le principe DRY (Do not Repeat Yourself)

par Bruno Orsier ([Site Web de Bruno Orsier](#))

Date de publication : 03/4/2008

Dernière mise à jour : 02/4/2008

Cet article présente le principe de programmation DRY (Do not Repeat Yourself - Ne vous Répétez pas), examine les principales causes de duplication de code ou plus généralement d'informations, et propose divers outils pour y remédier.

- I - Introduction
- II - Les bases du principe DRY
- III - Les causes de ces duplications
 - III-A - Les duplications imposées
 - III-B - Les duplications par inadvertance
 - III-C - Les duplications par impatience
 - III-D - Les duplications inter-développeurs
- IV - Exemples
 - IV-A - Synchronisation de code et de document
 - IV-B - Exemple de duplication par inadvertance
 - IV-C - Exemple de duplication inter-développeurs
 - IV-D - Exemple de duplication par impatience
- V - Conclusion
- Remerciements

I - Introduction

Les duplications de code et plus généralement d'informations posent de sérieux problèmes dans les développements de logiciels professionnels qui doivent être maintenus pendant une longue période : ces duplications rendent les futures évolutions plus risquées et peuvent causer des bugs très difficiles à identifier.


Un développeur professionnel devrait par conséquent être familier avec le principe DRY (Do Not Repeat Yourself), lequel se traduirait en français par "Ne vous répétez pas". Ce principe a été rendu populaire par le livre **The Pragmatic Programmer** de Andrew Hunt et David Thomas.

D'autre part ce principe est essentiel dans le Développement Dirigé par les Tests (TDD) car il constitue le principe directeur de la phase de remaniement de code (refactoring) qui vient juste après l'écriture de code permettant d'obtenir la barre verte (voir par exemple mon précédent [tutoriel](#)).


Enfin, il me semble que respecter ce principe est l'acte de **conception** le plus simple qu'un développeur débutant puisse apprendre, avant même de s'intéresser par exemple à certains patrons de conception.

Mais la principale difficulté de ce principe est qu'il paraît assez évident, si bien que l'on trouve peu d'informations complémentaires ou d'illustrations concrètes. Ce principe est souvent recommandé dans les blogs des développeurs ou des consultants, mais les commentaires restent la plupart du temps de haut niveau, comme par exemple celui de Karl Sequin récemment :

"Les duplications de code peuvent causer de forts maux de tête aux développeurs. Non seulement elles rendent le code plus difficile à changer (parce que vous devez trouver tous les endroits qui font la même chose), mais elles ont aussi le potentiel d'introduire de sérieux bugs et rendre la vie inutilement compliquée aux nouveaux développeurs. En suivant le principe Ne vous Répétez Pas durant toute la vie d'un système (histoires d'utilisateur, conception, codage, tests unitaires et documentation) vous arriverez à du code plus propre et plus maintenable. Gardez à l'esprit que le concept va plus loin que le copier/coller, et vise à éliminer les duplications de fonctionnalité/comportement sous toutes les formes. L'encapsulation des objets et du code très cohésif peut nous aider à réduire les duplications." **Karl Sequin**

 *Pour information, le texte original : "Code duplication can cause developers major headaches. They not only make it harder to change code (because you have to find all the places that do the same thing), but also have the potential to introduce serious bugs and make it unnecessarily hard for new developers to jump onboard. By following the Don't Repeat Yourself (DRY) principal throughout the lifetime of a system (user stories, design, code, unit tests and documentation) you'll end up with cleaner and more maintainable code. Keep in mind that the concept goes beyond copy-and-paste and aims at eliminating duplicate functionality/behavior in all forms. Object encapsulation and highly cohesive code can help us reduce duplication."*

De plus il y a peu d'informations en français sur la question. Il me semble donc utile de chercher à illustrer concrètement ce principe DRY, notamment avec des exemples de code. La première section de cet article reprend et reformule la présentation originale de Hunt et Thomas. La deuxième section reprend les causes de duplications également identifiées par Hunt et Thomas. Puis des exemples concrets basés sur mon expérience personnelle sont introduits dans la troisième section. J'ai choisi d'utiliser plusieurs langages de programmation (C#, Python, Delphi) afin d'insister sur le fait que ce principe est universel.

 *On pourrait peut-être traduire cet acronyme DRY par **SEC** en français : **Surtout Evitez les Copies***

II - Les bases du principe DRY

Le principe est souvent connu comme une interdiction de dupliquer du code. Mais en relisant la présentation faite dans **The Pragmatic Programmer** on se rend compte qu'il est beaucoup plus général. Les auteurs (Andrew Hunt et David Thomas) constatent qu'en tant que développeurs nous manipulons de la connaissance : nous travaillons sur sa collecte, son organisation, sa modélisation, sa maintenance, la faisons évoluer. Nous la documentons sous forme de spécifications, sous forme de commentaires dans le code, sous forme de diagrammes. Nous la rendons vivante sous forme de code exécutable, et nous l'utilisons pour les tests.

Toutefois cette connaissance que nous manipulons a une propriété remarquable : elle n'est pas stable. Elle change, souvent rapidement. Les besoins utilisateurs évoluent, des changements de réglementation doivent être pris en compte, le marché visé évolue lui aussi (par exemple à cause d'une nouvelle version produite par un concurrent, ou encore une fusion entre des concurrents qui change la donne). Les tests peuvent montrer qu'un algorithme n'est pas adapté. Notre compréhension de cette connaissance évolue au fur et à mesure de nos travaux, et nous souhaitons remanier le code, améliorer notre modèle métier.

Cette instabilité implique que nous passons une bonne partie de notre temps dans un mode de maintenance, à réorganiser cette connaissance. Il est illusoire de penser qu'une application entre en phase de maintenance après sa livraison : les développeurs sont plutôt constamment dans un mode de maintenance. Quand vous retouchez ou améliorez le code écrit hier, vous êtes déjà en train de le maintenir.

Réfléchir sur ce caractère omniprésent de la maintenance conduit à s'apercevoir qu'une grande partie de notre activité consiste à trouver et changer la représentation d'éléments de connaissances disséminés dans un logiciel. Malheureusement il est très facile de dupliquer de la connaissance dans les spécifications, processus, programmes, et tests que nous écrivons - et la maintenance peut devenir un cauchemar bien avant que l'application ne soit livrée.

Le respect du principe DRY devrait permettre d'éviter ce problème. Il n'est pas focalisé sur les duplications de code, mais sur les duplications de connaissances :

 **Principe DRY : chaque élément de connaissance doit avoir une représentation UNIQUE, NON AMBIGUË, OFFICIELLE dans un système**

Pour bien le comprendre on peut imaginer un cas où il n'est pas respecté : une chose est représentée à deux ou trois endroits différents. Si vous en modifiez un, vous devez vous rappeler de changer les autres. Pour Hunt et Thomas, **la question n'est pas de savoir si vous vous rappellerez, mais de savoir quand vous oublierez.**

III - Les causes de ces duplications

Andrew Hunt et Dave Thomas ont identifié quatre causes de duplications :


III-A - Les duplications imposées

Les développeurs ont le sentiment que la duplication leur est imposée par l'environnement. Par exemple une structure de classes doit refléter un schéma de base de données. Il y a toutefois souvent des solutions pour éviter la duplication de connaissances. Dans le cas de la base de données, il est possible de générer la structure des classes depuis le schéma de la base de données. Ainsi les solutions tournent souvent autour de l'utilisation d'un générateur de code à partir d'une représentation unique de connaissances. Par contre il est vital de rendre le processus actif, dans le sens où la génération doit pouvoir être refaite à volonté ; sinon l'on retomberait dans la duplication.

Les commentaires dans le code constituent un autre cas de duplication qui paraît imposée (par exemple par les normes de codage). Mais ce problème disparaît de lui-même si les commentaires sont de bonne qualité : ils doivent éviter de paraphraser le code, et doivent comporter des explications de haut niveau qu'il n'est pas possible de déduire de la simple lecture du code. Ainsi on évitera de devoir modifier les commentaires à chaque modification du code, et on évitera que les commentaires ne se désynchronisent du code.

Devinette : qu'est-ce qui est pire que l'absence de commentaires ? C'est la présence de commentaires obsolètes. Le code lui-même ne doit pas avoir besoin de commentaires pour être compris : il doit être auto-décrit, par l'utilisation de noms de variables et de fonctions très claires, et il doit avoir une complexité réduite.

Maintenir une parfaite synchronisation entre documentation et code est souvent difficile. Là aussi il faut autant que possible générer la documentation à partir du code.

 *Divers outils le permettent : pour Visual Studio, voir par exemple **GhostDoc** qui permet de générer automatiquement les entêtes de commentaires XML. Parfois il est possible de générer directement certains tests à partir d'un document de spécification. Voir notamment **FitNess** qui permet de rédiger des tests sous forme de tables dans les pages html d'un wiki, puis de les exécuter et de présenter les résultats dans le wiki. L'apparition récente d'un outil commercial comme **GreenPepper** (permettant de rédiger des "spécifications exécutables") est sans doute une indication de la pertinence des principes proposés initialement par FitNess.*


Enfin certains types de duplications imposées peuvent provenir directement du langage de programmation, qui parfois oblige à dupliquer les signatures de fonctions entre une partie interface et une partie implémentation (C++, Delphi). Il n'y a alors pas grand chose à faire, mais ici l'inconvénient est réduit du fait que le compilateur indiquera les éventuelles incohérences.

III-B - Les duplications par inadvertance

Les développeurs ne réalisent pas qu'ils sont en train de dupliquer de l'information. Il s'agit ici essentiellement d'erreurs dans la conception, quand deux classes contiennent le même élément d'information. D'autre part, une classe définit parfois des champs mutuellement dépendants ; dans ce dernier cas il est sans doute préférable de remplacer l'un des champs par une méthode réalisant le calcul à chaque accès.

III-C - Les duplications par impatience

Les développeurs dupliquent par paresse ou par facilité ou encore sous la pression de dates de livraison à respecter. Très souvent, nous préférons dupliquer une méthode, une classe, un fichier plutôt que de prendre le temps (et peut-être le courage) de factoriser proprement les éléments correspondants. Cette tendance naturelle est de plus encouragée par les contraintes de temps, et surtout par l'absence de tests automatisés. En effet, dupliquer permet de modifier une seule zone de code, alors que factoriser implique nécessairement de modifier au moins deux zones ; en l'absence de tests automatisés, le risque apparaît alors trop grand. Malheureusement le principal remède ici est l'autodiscipline et la volonté de passer plus de temps maintenant pour en économiser plus tard.

 *Un lecteur (**olsimare**) me signale très justement que ce type de duplication arrive dans le domaine des bases de données, et **au niveau des données elles-mêmes**. Par impatience, ou paresse, ou peur des risques, on duplique parfois des données au lieu de faire le travail plus difficile de restructuration qui aurait été nécessaire. Au fil du temps et des éventuels problèmes, la cohérence entre les données dupliquées devient de plus en plus difficile à maintenir.*

III-D - Les duplications inter-développeurs

Cela arrive quand des développeurs codent indépendamment la même fonctionnalité. Normalement une architecture de logiciel bien définie et claire devrait réduire ce risque, mais il y a toujours des fonctionnalités communes (comme des fonctions utilitaires) qui n'appartiennent pas clairement à tel ou tel élément de l'architecture. Il y a alors de grandes chances que ces utilitaires soient développés plusieurs fois. Le remède ici est de centraliser les utilitaires, de faciliter la communication entre développeurs (forums, wiki, ...), et de se forcer à lire le code de ses camarades. Le travail en duo (pair-programming) recommandé par XP apporte également des pistes de solutions.

IV - Exemples

IV-A - Synchronisation de code et de document

Quand on cherche à maintenir l'**unicité** des éléments de connaissances alors que l'on a besoin de cette connaissance sous différents formats, il faut utiliser un processus de génération automatique pour obtenir ces formats à partir de la représentation unique.

Les extraits de code que l'on inclut dans un livre ou un article sont un cas assez courant où un tel processus est très utile. Sans processus automatique, à chaque modification du code exemple, il faut penser à modifier le livre ou l'article, rechercher l'endroit où propager la modification, et éventuellement adapter la mise en page de ce que l'on vient de dupliquer. Tout cela est évidemment source d'erreurs, de travail supplémentaire, et de fatigue due au caractère inintéressant de ces opérations.

Récemment certains auteurs vont encore plus loin : le livre **xUnit Test Patterns** de Gerard Meszaros a d'abord existé sous la forme d'un site Web, et l'auteur a développé toute une série de scripts Ruby pour pouvoir générer son livre à partir du contenu du site Web. Par la suite, seul le livre a évolué, ce qui rend le site Web de plus en plus obsolète - c'est donc une violation du principe DRY, mais qui a été faite volontairement afin de favoriser la vente du livre !

Ici j'illustre ce même principe (à une échelle beaucoup plus réduite), en montrant comment je vais inclure des exemples de code dans cet article à l'aide d'un tel processus automatisé. Heureusement les articles publiés sur ce site sont des fichiers xml, ce qui facilite grandement leur manipulation par un programme séparé.

La première étape de mon processus est d'inclure une balise code dans mon fichier xml :

```
<code langage="python" autoinsert="F:\\DEV\\python\\remplace-code-dans-xml.py::signet-dvp-1">
--- le code sera inséré ici ---
</code>
```

La deuxième étape consiste à exécuter le script python sur mon document xml. Le script recherche simplement les noeuds "code", extrait les lignes de code situées entre les signets dans les fichiers indiqués par l'attribut "autoinsert", modifie les noeuds "code", et régénère le fichier xml (en ayant pris soin d'en faire une copie). Voici le script en question :

```
# Pour fonctionner sur des fichiers contenant des caractères accentués,
# ce script nécessite la présence des lignes
# import sys
# sys.setdefaultencoding('latin-1')
# dans le fichier Python/Lib/site-packages/sitecustomize.py
# reference : http://personalpages.tds.net/~kent37/blog/stories/14.html

from __future__ import with_statement
import os
import sys
import shutil
import xml.etree.ElementTree
from xml.etree.ElementTree import ElementTree

def traitementArticle(fichier):
    shutil.copyfile(fichier, nomFichierUnique(fichier))
    tree = ElementTree(None, fichier)
    map(insertionCodeDansBalisesXML, tree.getiterator('code'))
    tree.write(fichier, 'iso-8859-1')
    remplacementChr160DansFichier(fichier)
```

```
def insertionCodeDansBalisesXML(element):
    try:
        (fichier,signet)=element.attrib['autoinsert'].split('::')
        element.text = extractionLignesEntreSignets(fichier,signet)
    except:
        print "erreur en traitant: ", element.attrib
        print sys.exc_info()

def extractionLignesEntreSignets(fichier,signet):
    resultat=""
    with open(fichier) as f:
        for ligne in f:
            if ligne.find(signet)>=0:
                if resultat != "":
                    break
                else:
                    resultat = "\n"
                    continue
            if resultat != "":
                resultat += ligne
    if resultat == "":
        resultat = "signet " + signet + "non trouvé"
    return resultat

def fabriqueNomFichier(nom,index):
    return nom + "." + str(index)

def nomFichierUnique(fichier):
    index = 1
    while (os.path.exists(fabriqueNomFichier(fichier,index))):
        index += 1
    return fabriqueNomFichier(fichier,index)

# J'ai constaté que la représentation "&#160;" du caractère "espace insécable"
# de l'éditeur XML était remplacée par le caractère de code 160 durant la
# transformation. Pour l'instant je n'ai pas trouvé d'autre solution que de réouvrir
# le fichier pour remplacer le caractère 160 par la bonne représentation.
def remplacementChr160DansFichier(fichier):
    with open(fichier) as f:
        s = f.read().replace(chr(160),"&#160;")
    with open(fichier,'w') as f:
        f.write(s)

if __name__ == "__main__":
    traitementArticle('C:\\Article_Dvp\\documents\\principe_dry\\principe_dry.xml')
```


Comme l'attribut "autoinsert" n'est pas modifié dans ce processus, il est possible de recommencer autant de fois que nécessaire - ce qui est une propriété absolument indispensable pour éviter de retomber dans de futures duplications. Ma première idée était plutôt du type :

```
<code language="python">
autoinsert="F:\\DEV\\python\\replace-code-dans-xml.py::signet-dvp-1"
</code>
```

mais l'information sur "autoinsert" était perdue lors du remplacement par le code, et donc la propriété désirée n'était pas respectée.

Le script ci-dessus tente d'éviter des duplications : en particulier les noms de fonctions très explicites, les fonctions très courtes et de complexité réduite évitent de commenter le code. Ainsi les commentaires sont limités aux informations qu'il est impossible de déduire du code lui-même, et se focalisent sur le **Pourquoi** au lieu de dupliquer le **Comment**.

Toutefois la fonction `extractionLignesEntreSignets` n'est peut-être pas suffisamment facile à comprendre. Un commentaire pour expliquer ce qu'elle fait pourrait alors sembler utile : ignorer les lignes avant et après le signet, concaténer les lignes entre les deux occurrences du signet... Mais dans une optique DRY il est plutôt souhaitable de la réécrire afin de faciliter sa lecture, et sa future maintenance. Je laisse la réécriture en exercice pour les lecteurs !

 *Il se trouve que ce petit script Python illustre une des autres recommandations de **The Pragmatic Programmer** : **apprenez un langage de manipulation de texte**. Hunt et Thomas appellent ainsi les langages tels que Python, Perl, Ruby, qui permettent à un développeur d'automatiser diverses tâches répétitives et ainsi de pallier les limitations de son environnement de développement. J'adhère sans réserve à cette recommandation : si vous ne pratiquez pas de ces langages, apprenez-en un ! Vous pourriez être surpris de découvrir que cela vous rend service tous les jours. Et apprendre Python ou Ruby est l'affaire de quelques heures. Dans un contexte Windows/.NET, **PowerShell** est également une option.*

Pour moi le bilan de ce petit script est très positif : n'ayant encore jamais manipulé de XML avec Python, 1 heure de recherche et d'expérimentation m'a été nécessaire pour déterminer le cœur du script, à savoir l'obtention d'un itérateur sur toutes les balises "code". Cette heure est un bon investissement car manipuler des fichiers XML est un besoin fréquent pour un programmeur, et cela me resservira sans aucun doute. J'ai ensuite perdu beaucoup plus de temps à résoudre les questions d'encodages de caractères mentionnées dans les commentaires, mais le confort que m'apporte maintenant ce script vaut largement le temps passé. Mon seul regret est de ne pas avoir fait plus tôt ce petit effort, cela m'aurait évité du travail de synchronisation bien désagréable sur de précédents articles où mon seul outil était `Copier/Coller`.

IV-B - Exemple de duplication par inadvertance

Voici maintenant un exemple tiré d'un cas réel, où le non-respect du principe DRY a entraîné un défaut sérieux. A l'origine, un développeur A (moi-même en l'occurrence, il y a bientôt dix ans) doit afficher un signal, et doit le normaliser avant de l'afficher, c'est-à-dire ramener toutes les valeurs dans l'intervalle [0;100%]. Le développeur A programme donc le code suivant :

```
public class Afficheur
{
    private Donnees donnees;
    private Graphique graphique = new Graphique();

    public Afficheur(Donnees donnees)
    {
        this.donnees = donnees;
    }

    public void Affiche()
    {
        double pasX = 0 ;
        foreach (double donnee in donnees)
        {
            double valeur_normalisee = 100.0 * (donnee - donnees.Minimum) / (donnees.Maximum -
donnees.Minimum);
            pasX += 0.1;
            graphique.AddXY(pasX, valeur_normalisee);
        }
    }
}
```

Pour faire au plus simple et au plus rapide, le développeur A a fait le calcul de normalisation "au vol" et au dernier moment. Ce choix qui paraissait judicieux à l'époque du développement initial s'est révélé lourd de conséquences durant l'ajout de fonctionnalités supplémentaires (comme l'impression du graphique).

Pour information, ce code s'appuie sur le squelette de classes suivant (j'ai écrit le strict minimum pour parvenir à compiler) :

```
public class Graphique
{
    public void AddXY(double x, double y)
    {
        // ...
    }
}

public class Donnees : IEnumerable
{
    double minimum ;
    double maximum ;
    private ArrayList donneesInternes = new ArrayList();

    public Donnees (ArrayList donnees)
    {
        minimum = Double.MaxValue;
        maximum = - Double.MaxValue;
        foreach (double d in donnees)
        {
            minimum = Math.Min(minimum, d);
            maximum = Math.Max(maximum, d);
            donneesInternes.Add(d);
        }
    }

    public double this[int pos]
    {
        get { return (double)donneesInternes[pos]; }
    }
    public IEnumerator GetEnumerator()
    {
        return donneesInternes.GetEnumerator();
    }
    public double Maximum
    {
        get { return maximum; }
    }
    public double Minimum
    {
        get { return minimum; }
    }
}
```

Un an plus tard, un développeur B doit programmer un algorithme de calcul qui a besoin de plusieurs méthodes de normalisation. Malheureusement il ne touche pas au code de normalisation qui existe déjà dans la classe *Afficheur*. En effet,

- soit il ne connaît pas la classe *Afficheur*, et ne sait donc pas que cette méthode existe,
- soit il la connaît, mais il n'ose pas y toucher. Peut-être parce qu'il n'a pas le temps, ou encore parce que il n'ose pas toucher à du code existant (il n'a pas de tests automatisés qui lui permettraient de remanier le code en sécurité)
- soit il estime qu'il n'a pas à y toucher, car aucune des méthodes de normalisation qui l'intéressent n'est exactement la même que celle de la classe *Afficheur*

Quelle que soit la raison, le développeur B développe sa propre bibliothèque de plusieurs méthodes de normalisation et ne touche pas au code existant.

Deux ans plus tard, un développeur C doit réaliser l'impression de l'écran affiché initialement par le développeur A. Le moteur d'impression est totalement indépendant de la classe *Afficheur*, il n'est pas possible d'utiliser le même composant *Graphique*, et il faut en gros afficher les mêmes données sur un autre support (ce qui en soit est aussi une forme de duplication). Développeur C n'a donc pas accès au calcul de normalisation noyé dans la classe *Afficheur*, et s'oriente assez logiquement vers la bibliothèque de développeur B. Il y choisit une des méthodes fournies et l'utilise pour réaliser l'impression.

Hélas l'équivalent de la méthode contenue dans *Afficheur.Affiche* n'était pas dans la bibliothèque, et celle choisie donne à peu près les mêmes résultats sur certains jeux de données, en particulier ceux utilisés dans les tests manuels. Et ensuite pendant plusieurs années les résultats imprimés sont différents des résultats affichés. Mais cela passe inaperçu jusqu'à ce qu'un nouveau type de données chez un client rende la différence apparente.

Le développeur A écrit code de normalisation
directement dans GUI

```
Affiche()  
  
pasX = 0 ;  
(double donnee in donnees)  
  
le valeur_normalisee = 100.0 * (donnee - donnees.Minimum) / (  
    donnees.Maximum - donnees.Minimum);  
  
+= 0.1;  
Graphique.AddXY(pasX, valeur_normalisee);
```

Le développeur C réalise l'impression et
utilise une des fonctions de la
bibliothèque – pas la bonne car elle
n'est pas dans la bibliothèque

Un bug critique non détecté pendant plusieurs années
car les résultats sont souvent proches

Développeur B écrit bibliothèque
d'outils de normalisation
qui ne touche pas au
code existant
→ duplication

illustration d'un processus de duplication par inadvertance

La morale de cet exemple, simplifié mais réel, est qu'à force de ne pas traiter les duplications au fur et à mesure que l'on ajoute du code, on introduit de la confusion et des problèmes très difficiles à repérer. Cet exemple montre aussi pourquoi ces duplications ne sont pas traitées. D'une part, en l'absence de tests automatisés, un développeur préfère ne pas prendre de risque et préfère dupliquer plutôt que modifier du code existant. D'autre part, traiter les duplications peut exiger des remaniements qui apparaissent trop importants, et trop coûteux, au vu de l'état actuel de l'architecture du logiciel : c'est le cas si l'on avait voulu s'affranchir d'un support particulier pour pouvoir utiliser la même méthode d'affichage à l'écran que pour l'impression - cela remettait en cause l'architecture suivie jusque-là.

Ici on peut noter que si développeur A avait travaillé en TDD, le problème aurait pu être éliminé à la source. En effet travailler en TDD aurait obligé à extraire la méthode de normalisation de la classe *Afficheur* (afin de la rendre testable) et aurait donc eu pour conséquence bénéfique de la rendre disponible aux développeurs suivants. Développeur B aurait donc été incité à l'inclure dans sa bibliothèque, et développeur C aurait eu le choix de la bonne méthode.

Afin de terminer notre exemple, voici comment le problème aurait pu être traité en TDD. Tout d'abord voici un des tests qui servirait de point de départ :

```
[TestFixture]
public class TestNormalisation
{
    [Test]
    public void VerificationValeursAttendues()
    {
        ArrayList donnees_test = new ArrayList();
        donnees_test.Add(1.0);
        donnees_test.Add(2.0);

        Donnees donnees = new Donnees(donnees_test);

        Normalisateur norm = new Normalisateur(donnees);

        Assert.AreEqual(0.0, norm[0], 0.0001);
        Assert.AreEqual(1.0, norm[1], 0.0001);
    }
}
```

On voit que ce test fait appel à une nouvelle classe *Normalisateur*, qui traduit notre souci d'extraire la méthode de calcul afin de la rendre testable. Une autre option aurait été d'ajouter cette méthode de calcul directement à la classe *Donnees*, mais nous faisons volontairement ce choix de conception afin de limiter les responsabilités de la classe *Donnees*.

```
public class Normalisateur : IEnumerable
{
    private Donnees donnees;

    public Normalisateur(Donnees donnees)
    {
        this.donnees = donnees;
    }

    public double this[int pos]
    {
        get { return 100.0 * (donnees[pos]-donnees.Minimum)/(donnees.Maximum - donnees.Minimum); }
    }

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

```
}  
}
```

Bien sûr l'introduction de la classe *Normalisateur* a des conséquences sur notre *Afficheur*, mais ces conséquences sont positives car *Afficheur* dépend maintenant uniquement d'une interface *IEnumerable*. Nous avons donc réduit le couplage de nos classes.

```
public class AfficheurTDD  
{  
    private IEnumerable donnees;  
    private Graphique graphique = new Graphique();  
  
    public AfficheurTDD(IEnumerable donnees)  
    {  
        this.donnees = donnees;  
    }  
  
    public void Affiche()  
    {  
        double pasX = 0;  
        foreach (double donnee in donnees)  
        {  
            pasX += 0.1;  
            graphique.AddXY(pasX, donnee);  
        }  
    }  
}
```

IV-C - Exemple de duplication inter-développeurs

Il est fréquent que les développeurs redéveloppent le même code, le cas le plus évident étant celui de fonctions utilitaires nécessaires à de nombreux endroits dans une application. Prenons le cas de la fonction *GetTempPath* de l'API Windows. Elle n'est pas très pratique à appeler directement depuis un programme Delphi, car elle prend en paramètre un tableau de caractères, alors que l'on aimerait mieux une *string* Delphi, beaucoup plus pratique à manipuler. J'ai scanné les nombreux fichiers sources qui composent une de nos applications, et j'ai trouvé près de 50 appels à cette fonction *GetTempPath*; certains de ces appels sont noyés dans le code d'une autre fonction ou procédure, d'autres ont été encapsulés dans une fonction Delphi plus pratique. Ci-dessous je m'intéresse à des dernières fonctions, car j'en ai trouvé une grande variété.

Nous avons d'une part des fonctions qui travaillent avec un tableau de caractères interne de taille fixe, soit 1024 comme ci-dessous :

```
Function GetTempDir : String;  
Var  
    Buffer : Array[0..1023] Of Char;  
Begin  
    SetString(Result, Buffer, GetTempPath(SizeOf(Buffer), Buffer));  
End;
```

ou encore de taille *MAX_PATH+1* (constante Windows qui vaut 255) comme ci-dessous :

```
function GetTempDir: string;  
var  
    Buffer: array [0..MAX_PATH] of Char;  
begin  
    SetString(Result, Buffer, GetTempPath(SizeOf(Buffer), Buffer));  
end;
```

```
end;
```

Il y a encore d'autres variantes, qui travaillent avec un *PChar* au lieu d'un tableau de caractères, et qui allouent donc la mémoire de manière dynamique. Et qui prennent soin d'ajouter un caractère séparateur de chemin si besoin (ce qui est inutile d'après la documentation de *GetTempPath*) :

```
function GetPTempDirectory: string;
var
  pcRepertoire: PChar;
begin
  pcRepertoire := StrAlloc(255);
  try
    GetTempPath(254, pcRepertoire);
    Result := StrPas(pcRepertoire);
    if (Result[Length(Result)] <> '\') then
      Result := Result + '\';
  finally
    StrDispose(pcRepertoire);
  end;
end;
```

Voici une variante de la fonction ci-dessus :

```
Function DirectoryTemp : String;
Var
  tempdir : PChar;
Begin
  tempdir := Stralloc(255);
  Try
    If (GetTempPath(255, tempdir) > 0) Then
      result := String(tempdir)
    Else
      result := '';
  Finally
    StrDispose(tempdir);
  End;
End;
```

Toutefois aucune des fonctions ci-dessus ne capture une particularité de cette fonction *GetTempPath*, particularité que l'on découvre dans la documentation **MSDN** correspondante :

GetTempPath Function

Retrieves the path of the directory designated for temporary files.

```
DWORD WINAPI GetTempPath(
  __in  DWORD nBufferLength,
  __out LPTSTR lpBuffer
);
```

Parameters

nBufferLength

The size of the string buffer identified by lpBuffer, in TCHARs.

lpBuffer

A pointer to a string buffer that receives the null-terminated string specifying the temporary file path. The returned string ends with a backslash, for example, C:\TEMP\.

Return Value

If the function succeeds, the return value is the length, in TCHARs, of the string copied to lpBuffer, not including the terminating null character. If the return value is greater than nBufferLength, the return value is the length, in TCHARs, of the buffer required to hold the path.

La particularité en question est que deux appels successifs sont nécessaires afin de gérer le cas où le tableau de caractères passé en paramètre est de taille insuffisante. Certaines implémentations tentent bien de prendre cela en compte, mais pas forcément de manière bien adroite comme c'est le cas ci-dessous :

```
//Get the temp folder
TempStrLength := 0;
repeat
  TempStrLength := TempStrLength + 10;
  tempstr := StrAlloc(TempStrLength);
  try
    ret := GetTempPath(TempStrLength, tempstr);
    TempPath := tempstr;
  finally
    StrDispose(tempstr);
  end;
until (ret < TempStrLength);
```

Pour finir avec les exemples de code, voici une implémentation correcte que l'on trouve dans la bibliothèque **JEDI Code Library** :

```
function GetWindowsTempFolder: string;
var
  Required: Cardinal;
begin
  Result := '';
  Required := GetTempPath(0, nil);
  if Required <> 0 then
  begin
    SetLength(Result, Required);
    GetTempPath(Required, PChar(Result));
    StrResetLength(Result);
    Result := PathRemoveSeparator(Result);
  end;
end;
```

Que conclure de cette débauche de duplication de code et d'efforts ?

- C'est un problème courant : j'ai trouvé de tels exemples dans notre propre code mais aussi dans beaucoup de composants achetés.
- Ces duplications sont d'autant plus néfastes qu'elles concernent la duplication de code incorrect ne prenant pas bien en compte les particularités de *GetTempPath*.
- Si un jour il est nécessaire de corriger les tailles des tableaux de caractères définis dans ces fonctions, il sera nécessaire de recompiler pratiquement tous les EXE et DLL qui composent l'application. Il est dommage qu'une fonction *GetWindowsTempFolder* n'ait pas été définie de manière centralisée dans une DLL.
- De même si l'on souhaite un jour utiliser un dossier temporaire propre à l'application, cela impacterait également tous les composants de l'application. La duplication intempestive réduit donc les possibilités d'évolution.

Il n'est hélas pas très simple de mettre en place une politique de prévention de telles duplications. En effet dans ce cas précis nombre de développeurs trouveront sans doute plus rapide de recoder une fonction autour de *GetTempPath* plutôt que de faire des recherches pour savoir si une fonction utilitaire similaire a déjà été développée (mais il la recoderont probablement incorrectement comme les exemples ci-dessus le montrent !). En ce qui me concerne **je**

Les exemples ci-dessus pourraient laisser penser que des duplications n'apparaissent que dans des centaines de milliers de lignes de code difficiles. Il n'en est rien, il est courant d'introduire des duplications dans quelques lignes de code comme le montre l'exemple ci-dessous.

```
procedure TrouverLeFichierLePlusAncien(const nomDossier: string);
var
  search_record: TSearchRec;
  chemin_de_recherche: string;

  plus_ancienne_date: _FILETIME;
  plus_ancien_fichier: string;
  resultat_recherche: Integer;
  fichier_courant: string;
begin
  plus_ancienne_date.dwLowDateTime := MaxLongInt;
  plus_ancienne_date.dwHighDateTime := MaxLongInt;

  fichier_courant := '';
  plus_ancien_fichier := '';

  chemin_de_recherche := IncludeTrailingPathDelimiter(nomDossier) + '*.xml';

  resultat_recherche := sysutils.FindFirst(chemin_de_recherche, faAnyFile, search_record);
  while (resultat_recherche = 0) do
  begin
    if VerifierFichierNonReadOnly(search_record) then
    begin
      fichier_courant := IncludeTrailingPathDelimiter(nomDossier) + search_record.Name;

      if not VerifierFichierDejaOuvert(fichier_courant) then
        CalculerFichierPlusAncien(search_record.FindData.ftCreationTime, fichier_courant,
        plus_ancienne_date, plus_ancien_fichier);
      end;
      resultat_recherche := sysutils.FindNext(search_record);
    end;

    sysutils.FindClose(search_record);

    fichier_courant := plus_ancien_fichier;
    if (fichier_courant <> '')
      //signet-refaire-la-verification
      and not VerifierFichierDejaOuvert(fichier_courant) and VerifierFichierNonReadOnly(search_record)
      //signet-refaire-la-verification
    then begin
      // ... traitement du fichier
    end;
  end;
end;
```

La fonction ci-dessus recherche le fichier `.xml` le plus ancien dans un dossier, en prenant soin d'ignorer le fichier en read-only et les fichiers déjà ouverts par un autre processus (sinon le traitement ultérieur échouerait).

Dans la maintenance de cette fonction, il a été jugé nécessaire de refaire ces deux vérifications au dernier moment, juste avant le traitement, car des utilisateurs avaient mis en évidence des cas où le statut d'un fichier pouvait changer entre la boucle de recherche et le traitement. Aussi le code suivant a été ajouté dans un deuxième temps :

```
and not VerifierFichierDejaOuvert(fichier_courant) and VerifierFichierNonReadOnly(search_record)
```

Quel rapport avec le sujet du principe DRY ? Eh bien ce code ajouté ne respecte pas le principe : il **duplique** le code des vérifications faites dans la boucle. Et de plus il duplique avec une erreur difficile à identifier. En effet, lors du Copier/Coller, l'argument de la fonction `VerifierFichierNonReadOnly` n'a pas été modifié, alors qu'il n'est plus valide

à ce stade (un *FindClose* ayant été fait juste avant). La vérification finale n'est donc pas faite sur le bon fichier, mais cela ne se remarque que dans des cas particuliers !

Comment aurait-il fallu traiter ce cas ?

- Une première piste serait de factoriser les deux vérifications en une seule fonction - dans ce cas le risque de l'appeler avec un mauvais argument aurait été réduit, ou en tout cas le problème aurait pu être détecté rapidement
- Une autre piste consisterait à supprimer radicalement la duplication, et à considérer que la vérification faite dans la boucle n'a plus guère de sens à cause de la vérification finale. On aurait donc pu laisser uniquement la vérification finale, ce qui aurait pu attirer l'attention sur l'utilisation abusive de *search_record*.

En tout cas, le fait de **dupliquer** aurait dû déclencher un signal d'alarme ! Cela n'a pas été le cas et cette duplication a causé quelques désagréments à un client.

V - Conclusion

Avec mon expérience de développement et de maintenance d'un logiciel pendant près de 10 ans, je constate assez fréquemment les problèmes difficiles posés par la duplication de code ou de connaissances. En réalité, plus que la duplication elle-même, c'est plutôt la duplication erronée ou inconsistante qui pose problème en pratique. Soit la duplication était erronée dès le départ, soit le code dupliqué évolue de façon inconsistante à plusieurs endroits. Cela rend d'autant plus difficile la recherche automatisée de duplications. Les rares outils de recherche de duplication comme **Simian - Similarity Analyser** (commercial) ou **CPD** (gratuit) semblent surtout efficaces pour identifier le code dupliqué à l'identique, et butent sur les duplications inexactes.

Hélas il n'y a pas de solution bien simple, et la première action possible est probablement la sensibilisation des développeurs aux risques cachés derrière la duplication, afin qu'ils prennent le temps de réfléchir au moment où ils font Copier/Coller. Avec cet article j'espère avoir contribué à cette sensibilisation. J'ai également indiqué divers outils qui peuvent faciliter la vie des développeurs soucieux de respecter le principe DRY.

Remerciements

Merci à Luc Jeanniard, Emmanuel Etasse et Arnaud Pierrel pour leur relecture de cet article, et à **Nip** et **olsimare** pour leurs corrections, suggestions et encouragements.

