Formation C# – Delphi.NET – Delphi Win32
Développement & Sous-traitance

Migration de code Delphi Win32 vers .NET – Partie 1

Si le développement de nouvelles applications sous .NET ne demandent qu'à suivre la logique du langage et de la bibliothèque d'affichage choisis, reprendre un code VCL Win32 pour en assurer la migration sous VCL.NET peut présenter des écueils malgré l'effort important que Borland a fourni pour assurer une

compatibilité ascendante proche de la perfection. Savoir comment procéder est ici tout aussi essentiel que de savoir où se situeront les zones délicates pour les isoler rapidement. Le présent article ainsi que le suivant vous montreront à la fois la méthodes à utiliser et les pièges à éviter pour réussir vos migrations.

Le portage des applications Win32

Le portage d'une application d'une plate-forme à une autre n'est jamais dépourvu de surprises, bonnes et mauvaises, qui rendent souvent la tâches ardue et hasardeuse en terme de budget (temps et financier). Savoir ce qu'il faut migrer et comment effectuer cette migration est d'une importance au mois aussi grande que de savoir comment, techniquement, coder ou re-coder telle ou telle partie d'une application.

L'intérêt de passer en VCL.NET

La question mérite d'être posée et cette section sur le portage des applications semble un bon endroit pour y répondre… En effet, pour une application existante, vaut-il mieux la passer en VCL .NET ou bien la refaire en Windows Forms ?

En dehors du gain de temps évident qui prêche en faveur de la VCL .NET, le débat reste ouvert. Un développeur Delphi qui utilisera Delphi .NET et la VCL .NET ne perdra aucun temps en formation lourde sur un nouveau langage et une nouvelle bibliothèque d'affichage et cela est essentiel. D'un point de vue commercial, il est toujours préférable de sortir un produit perfectible en temps et en heure plutôt qu'une perfection (jamais atteinte) mais après la concurrence…

Si on met de côté ces aspects liés au temps d'apprentissage, le choix de la VCL .NET n'est pas « pire » techniquement que Windows Forms, en tout cas en première approximation. En effet, le futur de l'affichage sous .NET est XAML et Windows Vista sort en 2006. A ce moment là Windows Forms sera-t-elle une librairie d'affichage « has been » ? Cela est difficile à dire mais tout porte à croire qu'à terme XAML remplacera totalement Windows Forms. Cette dernière aura peut-être encore un avenir grâce aux unités mobiles et au Compact Framework ou bien grâce à son portage sous MONO.

Dès lors choisir VCL.NET est tout aussi « risqué » que de choisir Windows Forms.

Reste la cohérence des développements. En effet, VCL.NET n'est utilisable que sous Delphi.NET alors que Windows Forms est utilisable par tous les langages de la plate-forme, dont Delphi.NET.

De fait, celui qui doit créer des composants visuels aur-t-il intérêt à le faire sous WF plutôt que sous VCL.NET afin d'élargir considérablement sa base d'utilisateurs potentiels sans se priver des utilisateurs Delphi.

Celui qui doit aussi intégrer des travaux venant d'autres développeurs ou qui doit créer des assemblages utilisables dans d'autres langages aura-t-il tout intérêt à choisir Windows Forms plutôt que VCL.NET.

Le choix Windows Forms ou VCL.NET n'est pas sans conséquence et nous vous laissons tirer vos conclusions de ces réflexions et choisir la meilleure solution pour vos développements.

Portage de code

Toutes fonctions, bibliothèques de fonctions et autres codes, qui ne font aucun appel à la plate-forme Windows ni à l'interface sont portables dans les meilleures conditions qu'on puisse imaginer. Toutefois, il convient de bien faire attention à la nouvelle donne introduite par le passage à .NET. Toutes les nouveautés que nous avons vues jusqu'ici à propos du langage s'appliquent bien entendu à tout code migré de Win32 à .NET autant qu'au code créé *ex nihilo*.

Ce qui ne réclame pas de changement

Les éléments de langage suivants restent globalement identiques à ce qu'ils étaient, au moins dans leur fonctionnement et ne réclament donc, à première vue, aucun changement :

- les chaînes et les tableaux (types `string` et `array`) ;
- les enregistrements (type `record` mais ils supportent de nombreuses choses nouvelles) ;
- la déclaration des classes ;
- la déclaration et l'utilisation des interfaces ;

- les propriétés et les événements (voir la nouvelle notation pour les événements multicast) ;
- les ensembles ;
- les fichiers texte (type `TextFile`) ;
- les composants (en dehors des modifications que nous verrons plus loin) ;
- la gestion des flux (types dérivés de `TStream`).

Ce qui posera des problèmes

Tout ce qui n'est pas considéré comme sécurisé posera des problèmes de migration sous .NET et réclamera très certainement une réécriture partielle ou totale du code impliqué.

Pour vous aider à débusquer les portions de code à problème, n'oubliez par que le compilateur de Delphi 7 ainsi que la personnalité Win32 Delphi de BDS savent générer des avertissements spécifiquement liés à ces problèmes de portage. N'hésitez pas vérifier votre code sous Delphi Win32 et à le corriger sous cet environnement avant de le migrer sous Delphi .NET… Il est bien plus simple de changer du code dans un environnement où il fonctionne encore que d'avoir à modifier de nombreuses choses sous Delphi .NET avant d'espérer pouvoir faire une première compilation. Les sessions de modification trop longues dans le temps sont fastidieuses et sources d'erreurs.

Pour activer les avertissements .NET sous Delphi Win32 (ils sont désactivés par défaut), vous pouvez passer par les options d'environnement dans l'IDE ou ajouter au début de chaque unité les commutateurs suivants :

- `{$WARN UNSAFE_TYPE ON}`
- `{$WARN UNSAFE_CODE ON}`
- `{$WARN UNSAFE_CAST ON}`.

Les types non sécurisés suivants devront être changés :

- `Pchar`, `PWideChar` et `PAnsiChar` ;
- tous les pointeurs non typés ;
- tous les paramètres de type var et out non typés ;
- les fichiers déclarés `File Of <type>` ;

- les vieux réels Turbo Pascal déclarés `Real48` ;
- les enregistrements (`record`) utilisant des parties variables.

Toutes les parties de code contenant des tournures mettant en œuvre les instructions, les fonctions ou les éléments de syntaxe suivants devront être réécrits autrement :

- le code utilisant des variables absolues déclarées avec `absolute` ;
- les procédures standard impliquant explicitement des emplacements mémoire précis comme `Addr()`, `Ptr()`, `Hi()`, `Lo()` et `Swap()` ;
- les anciennes routines de lecture et d'écriture sur les fichiers non typés `BlockRead()` et `BlockWrite()` ;
- l'utilisation de `Fail` dans les constructeurs de classe ;
- les opération d'allocation et de libération de mémoire `GetMem()`, `FreeMem()` et `ReallocMem()` ;
- les opérations de remplissage ou de déplacement de contenu mémoire utilisant `Fillchar()` et `Move()` ;
- le code utilisant l'ancien type `Object` hérité du Turbo Pascal ;
- l'assembleur en ligne et le code utilisant `asm` ;
- les unités utilisant `ExitProc` ;
- l'opérateur d'adresse mémoire `@`.

Les transtypages suivants devront être supprimés et réécrits :

- Transtyper un objet dans un type qui n'est pas un ancêtre ou un descendant du type d'instance n'est plus possible (et c'est une bonne chose !) .
- Transtyper un enregistrement (`record`) en n'importe quoi d'autre.

De manière générale, Delphi .NET considère un transtypage comme un opérateur `as`. De fait, le transtypage ne réussit qu'à la seule condition que la donnée transtypée soit réellement du type indiqué. Le transtypage sauvage n'est donc plus autorisé.

Bien que ne posant pas directement un problème, vous noterez que sous Win32, les valeurs flottantes qui possèdent moins de quatre

chiffres après la virgule se transforment en type `Currency` lorsqu'elles sont affectées à un `Variant`. Sous Delphi .NET, le type devient un `Double`. Cela ne change donc rien aux calculs mais peut poser des problèmes si le type du contenu du `Variant` est testé dans le code. Dès lors, ce dernier ne trouvera plus le type `Currency` auquel il pouvait s'attendre, ce qui peut être la source d'un bogue très difficile à découvrir.

Les applications VCL et celles qui utilisent des accès aux bases de données peuvent également être une source de perte de temps. Il est en effet nécessaire d'ajouter l'attribut personnalisé `[STAThread]` avant l'instruction `begin` du fichier projet (`DPR`) afin de spécifier le modèle de threading `STA` (Single Threading Apartment). Si vous oubliez cette modification, vous obtiendrez des erreurs à l'exécution dont il sera difficile de comprendre la raison.

Les grandes équivalences

Le framework .NET possède de nombreuses facettes qui ressemblent tellement à celles de la VCL qu'il est facile de les substituer les unes aux autres (notamment grâce au travail effectué dans Delphi .NET pour facilité cette similitude). Le tableau 1 suivant montre les équivalences les plus marquantes.

Tableau 1

Équivalences de types Win32 / .Net

DELPHI WIN32	DELPHI .NET
`TObject`	`System.Object`
`Variant`	`Variant` défini comme `System.ValueType`
`Exception`	`System.Exception`
`TComponent`	`System.ComponentModel.Component`
`String`	`System.String`
`Record`	`System.ValueType` (nouvelle syntaxe)
`AnsiChar`	Simulé (`Borland.Delphi.System.AnsiChar`)
`ShortString`	Simulé (`Borland.Delphi.System.ShortString`)
`AnsiString`	Simulé (`Borland.Delphi.System.AnsiString`)
`WideString`	`System.String`
IHM	`System.Windows.Forms` et la VCL .NET

À noter : le type `string` de .NET n'est pas efficace lorsque la chaîne est modifiée de nombreuses fois. Pour les chaînes de ce type, il est préférable d'utiliser le type `StringBuilder` (appartenant à l'espace de nom `System.Text`). En effet, le type `string` est immuable et toute modification de la chaîne oblige à la création d'une nouvelle copie de la chaîne. L'ancienne sera certes détruite automatiquement par le ramasse-miettes de façon transparente mais cela pénalise beaucoup le temps d'exécution. De son côté, la classe `StringBuilder` alloue un espace de mémoire non managé et gère celui-ci de façon autonome. Il est facile de la convertir en string à la fin du travail sur la chaîne, le coût de la transformation étant minime en passant par la méthode `ToString` (les données de la chaîne ne sont pas copiées à nouveau).

Sous .NET, les types primitifs de données sont spécifiques à la plate-forme. On y retrouve néanmoins tous ceux que Delphi proposait déjà, même si dans certains cas, des adaptations ont été réalisées. Le tableau 2 liste les équivalences pour les types de base.

Tableau 2

Équivalences des types de base

DELPHI .NET	PLATE-FORME .NET
CARACTERES	
`Char`	`System.Char`
`WideChar`	`System.Char`
LOGIQUES	
`Boolean`	`System.Boolean`
ENTIERS	
`Byte`	`System.Byte`
`Shortint`	`System.SByte`
`Smallint`	`System.Int16`
`Integer`	`System.Int32`
`Word`	`System.UInt16`
`LongWord`	`System.UInt32`
`Cardinal`	`System.UInt32`
`Int64`	`System.Int64`
FLOTTANTS	
`Single`	`System.Single`
`Double`	`System.Double`
`Real`	System.Double
`Decimal`	System.Decimal
`Currency`	Il n'existe pas (voir notes après le tableau).
`Extended`	Il n'existe pas (voir notes après le tableau).
`Comp`	Deprecated, alias vers `System.Int64`

Les `Extended` ne sont pas présents dans la définition du CLR, mais Borland fournit une simulation de cette classe dans l'espace de nom `Borland.Delphi.System`. En réalité, ce type devient un simple alias vers le type `Double` du CLR. Sa déclaration dans les sources de Borland est :

```
type
  Extended = type Double;     // 80 bit reals are unique to the Intel x86 architecture
```

Le type `Currency` n'existe pas non plus dans le CLR et Borland en fournit aussi une simulation dans `Borland.Delphi.System`. Toutefois, à la différence des `Extended` qui ne sont que des alias, les `Currency` font l'objet d'une réelle simulation. Ils sont définis comme type par valeur (`record`) qui surcharge tous les opérateurs nécessaires à l'utilisation des instances de ce type.

Changement de nom des packages

Les packages utilisés par le code Win32 et se trouvant référencés dans la partie `Requires` d'un projet doivent être renommés, les noms simples Win32 devant être transformés en espace de nom .NET. L'ouverture d'un projet Win32 en mode .NET enclenche l'expert d'importation de Delphi qui effectue déjà un certain nombre de transformations automatiques. La grille de correspondance proposée par le tableau 3 permet de contrôler ou compléter votre code source.

Tableau 3

Équivalences des noms de package

WIN32	.NET
rtl	Borland.Delphi et Borland.VclRtl
vcl	Borland.Vcl
vclx	Borland.VclX
dbrtl	Borland.VclDbRtl
bdertl	Borland.VclBdeRtl
vcldb	Borland.VclDbCtrls
dbexpress	Borland.VclDbExpress
dbxcds	Borland.VclDbxCds
dsnap	Borland.VclDSnap
dsnapcon	Borland.VclDSnapCon
vclactnband	Borland.VclActnBand
IBXpress	Borland.VclIBXpress

Les packages peuvent être installés dans l'EDI en utilisant le menu Composant / Composants .NET installés / Composants VCL .NET, ou en cliquant avec le bouton droit de la souris dans le gestionnaire de projet et en choisissant la commande Installer.

Packages liés implicitement

Lors de la compilation d'une application Win32, le compilateur de Delphi .NET peut retourner un avertissement du type « Unité liée implicitement ». Considérez ces avertissements comme des erreurs fatales. Ces avertissements sont de toute façon considérés comme des erreurs fatales de compilation sous .NET. Autant les prendre en compte dès le départ.

Corriger la situation n'est généralement pas compliqué. Il suffit de s'assurer que la clause `Requires` liste bien tous les packages nécessaires et que la clause `Contains` donne la liste exhaustive de toutes les unités du package.

Par exemple, si Delphi .NET indique que `Borland.Vcl.Classes` est lié implicitement, vous devrez ajouter `Borland.VclRtl` à la clause `Requires`. Si le compilateur vous dit que `Borland.Vcl.Controls` est lié implicitement, c'est qu'il manque `Borland.Vcl`.

Compilation avec ou sans package d'exécution

La gestion du choix avec ou sans package d'exécution s'effectue de façon extrêmement simple sous Delphi .NET. Pour lier votre application à un package d'exécution (que vous aurez à déployer aussi), il suffit que son nom soit ajouté dans la catégorie Références du nouveau gestionnaire de projet. Pour ajouter un package, utilisez le menu Projet>Ajouter une référence. Une fois le package ajouté un clic droit sur son nom permet de choisir dans le menu local si on désire en faire une copie locale ou non (en vue de son déploiement) et son on souhaite ou non lié le package dynamiquement à l'exécutable.

Ceci n'est valable qu'avec des packages ou des unités générés par Delphi, les unités du framework ne peuvent jamais être liées à l'exécutable.

Que faire du code qui ne passe pas ?

Si vous possédez un code Delphi qui ne peut pas être migré facilement vers .NET et que vous n'avez pas le temps ou le budget pour envisager une réécriture totale, une solution d'attente (mais non définitive) consiste à transformer le code à problème en DLL Win32 puis à utiliser la technique de P/Invoke pour appeler ce code depuis vos applications Delphi .NET.

Portage d'applications VCL

Un certain nombre de composants de la VCL n'existent pas sous .NET. En dehors des composants tiers dont le portage dépend de leur nature et des intentions de leurs éditeurs, la VCL .NET ne supporte plus un certain nombre de techniques (comme la création de serveurs Midas, WebBroker pour faire des CGI et des ISAPI…).

En dehors de ces techniques qui sont majoritairement absentes parce qu'il existe des équivalents .NET bien plus efficace, tous les composants de la palette Standard sont présents. Il manque dans la palette Supplément les composants `TeeChart` mais il existe une version light gratuite téléchargeable sur le site www.steema.com. Les composants de la palette Win32 ont tous été repris, et même ceux de la palette Win3.1 assurant la compatibilité avec Delphi 1. La palette Système est presque complète, seuls manquent les composants liés au DDE et le `TOleContainer`. La palette Dialogues est elle aussi complète. On retrouve sous Delphi .NET une équivalence pour dbExpress, DataSnap, dbGo, Interbase Express et Indy.

Delphi .NET sait ouvrir les fichiers projets DPR des versions précédentes et fournit un premier niveau de traduction automatique. Lorsqu'on enregistre une application Win32, Delphi .NET crée un fichier projet dont l'extension est `bdsproj`, et écrit aussi dans le fichier `cfg` de ce dernier. Ces modifications n'interfèrent pas avec l'EDI de Delphi 7 ni de la personnalité Win32 des versions suivantes qui peut rouvrir le projet et le compiler à nouveau.

Lors de l'importation d'un projet Win32, Delphi .NET ajoute une référence au modèle de composant dans la clause `Uses` de la partie Interface de l'unité. Il s'agit de `System.ComponentModel` du framework. Bien entendu, cela ne sera pas reconnu par Delphi Win32 si vous rouvrez le projet. Si ce dernier doit pouvoir continuer à être compilé par les deux environnements, il vous faudra alors modifier cet ajout automatique en utilisant un test de commutateur comme suit :

```
uses
Windows, Classes, Graphics, Forms, Controls,
{$IFDEF CLR} System.ComponentModel, {$ENDIF} StdCtrls;
```

Normalement, vous devrez effectuer cette modification dans toutes les unités de votre projet.

Vous noterez qu'à la compilation, les unités .NET se transforment en `.dcuil` et qu'il n'y a donc aucun risque de mélange possible avec les unités compilées sous Win32 qui portent l'extension `.dcu`.

Parmi les petites choses pouvant générer des erreurs importantes ou obliger à la réécriture de code, il est bon de savoir que les méthodes `BeforeDestruction` et `AfterConstruction`, tout comme le champ `OldCreateOrder`, ne sont pas portés dans la VCL .NET. Tout code reposant sur ces éléments doit donc être revu. La VCL .NET au niveau de `OnCreate` et `OnDestroy` se comporte comme si `OldCreateOrder` était toujours égal à `True`.

Dans le même ordre d'idée, il est essentiel de savoir que les destructeurs ne sont pas forcément appelés sous .NET. De fait, comme `OnDestroy` dans une fiche `TForm` est désormais systématiquement appelé par le destructeur, il n'y a aucune garantie que le code soit exécuté. Le ramasse-miettes peut en effet fort bien libérer la mémoire sans que le destructeur ne soit appelé. Cela peut être contourné en appelant explicitement la méthode `Free`.

Utiliser la compilation conditionnelle pour maintenir la compatibilité

Delphi Win32 définit les commutateur `MSWINDOWS` et `WIN32` alors que Delphi .NET définit `CLR`, `CIL` et `MANAGEDCODE`. Kylix, pour sa part, définit le commutateur `LINUX`. En utilisant habilement ces commutateurs et la compilation conditionnelle, il est tout à fait possible de maintenir du code Delphi multi plates-formes. En voici un exemple trivial en mode console :

```
project MultiPlatform;
{$APPTYPE CONSOLE}
begin
{$IFDEF CLR} // Delphi .NET
    writeln('Bienvenue sous .NET !');
{$ENDIF}
{$IFDEF WIN32} // Delphi Win32
    writeln('Bienvenue sous Win32 !');
{$ENDIF}
{$IFDEF LINUX} // Kylix
    writeln('Bienvenue sous Linux !');
{$ENDIF}
end.
```

Si vous devez lier des fiches pour des applications impliquant l'IHM, n'oubliez pas que les extensions de fichiers sont différentes entre les trois environnements :

- .DFM sous Win32 ;
- .XFM sous CLX ;

- .NFM sous VCL.NET
- et aucune sous Windows Forms puisque le design de la fiche est inclus dans le code de l'unité et non dans un fichier séparé.

Reprendre une application VCL Win32

Les fiches VCL sont portables à peu près à 100% pour les composants standards mais le code peut, lui, réserver quelques surprises, ce qui impose de bien comprendre les modifications du langage.

De façon résumée, pour réussir une migration, utilisez la technique que nous préconisons à savoir la mise en route des commutateurs d'avertissements sous Delphi Win21 et *mutando, mutandis* vérifiez et modifiez votre code sous cet environnement. Lorsque vous n'obtenez plus un seul avertissement et que vous avez échangé tous les composants qui n'existent pas sous VCL .NET par des équivalents ou des `Frames`, à ce moment seulement vous pouvez envisager de charger le code sous Delphi .NET. À la section suivante, nous allons voir en détail toutes ces étapes du passage d'une application Delphi Win32 à Delphi .NET.

Tag mutant !

Tous les composants repris dans la VCL .NET sont en tout point similaires à ceux de la VCL Win32 excepté une propriété : `Tag`. En effet, Borland s'est bien rendu compte que certains utilisent la propriété `Tag` pour y stocker de tout sauf des entiers. Ne vous sentez pas visé, il n'y a pas de webcam cachée dans votre bureau ! Comme sous .NET, la pratique du transtypage sauvage n'est plus autorisée (ou ne se comporte plus comme avant, voir le boxing), il a été nécessaire d'officialiser le mariage parfois contre nature entre un `Longint` et un objet. De fait, la propriété `Tag` est aujourd'hui un `Variant`, ce qui permet d'y mettre à peu près n'importe quoi. La pratique n'a donc pas changé, mais elle se fait désormais en toute tranquillité. Le code spaghetti est désormais une pratique approuvée, mais n'en abusez pas au risque que ces pâtes-là ne vous restent sur l'estomac !

Migrer une application Win32

Comme nous n'allons pas reprendre tous les composants des palettes de la VCL .NET et consacrer tout un article aux choses que vous connaissez déjà ou pour lesquelles l'aide de l'EDI est plus que suffisante, nous allons plutôt décomposer les phases d'un portage d'application Win32 vers .NET, sujet autrement plus délicat et source d'inquiétude pour bon nombre de développeurs.

L'application de test

Il serait malhonnête de dire que nous avons choisi une application au hasard pour cette démonstration. En effet, les applications réelles dont nous disposons sont soit trop lourdes, soit trop techniques. Les premières nécessiteraient bien plus que quelques pages pour passer en revue tout le cycle de portage. Les secondes sont des utilitaires système ou base de données et en expliquer le fonctionnement ou le pourquoi serait plus long que la migration.

Nous avons donc choisi de construire une application simple à partir d'éléments existants. Il y a donc bien du code Delphi Win32 réel derrière, même du vieux code Delphi 5, seul l'habillage a été recomposé. Dans ce dessein, nous avons veillé à ce que l'application utilise des éléments bien spécifiques à Win32 et à ce qu'elle mime un comportement tout à fait standard d'application réelle.

Code complet
N'oubliez pas que le code complet des exemples du livre se trouve sur le CD-Rom fourni avec le livre. Reportez-vous aux annexes pour les détails d'installation.

Principe de fonctionnement

Cette application affiche une grille de données puisant sa source dans un `TClientDataset` ouvrant un fichier binaire (.cds). Depuis l'écran principal, il est possible d'imprimer la grille en HTML, soit en mode prévisualisation qui ouvre le navigateur Internet, soit en mode impression directe (qui passe aussi par le navigateur HTML installé sur la machine).

Les composants et le code utilisés

Dans cette application, nous avons utilisé des composants standard répartis sur plusieurs palettes en créant une situation réelle minimaliste par la présence des éléments suivants :

- une grille de données ;
- une liste d'action ;
- un menu principal ;
- une barre d'outils avec ses boutons ;
- un objet de la palette système (*timer*) ;

- une unité de code supplémentaire définissant la classe d'impression en HTML ;
- des fonctions appelant les `ShellApi` pour appeler le navigateur Internet ;
- un composant non standard extrait de la JVCL[1] permettant de choisir la couleur de la première colonne de la grille.

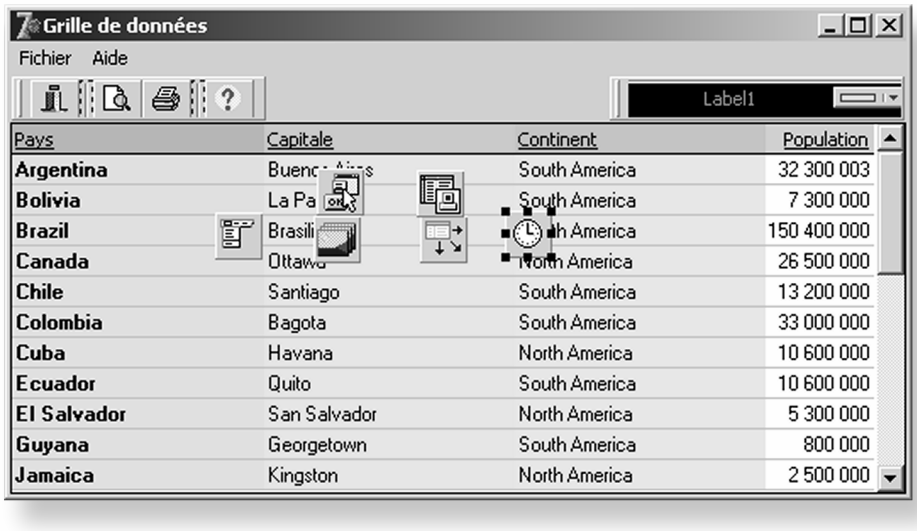


Figure 1

L'application de test à migrer sous IDE Delphi 7

La figure 1 montre l'écran principal de l'application sous IDE de Delphi 7. Vous pouvez y reconnaître les composants visuels évoqués précédemment ainsi que les composants non visuels suivants :

- le menu principal ;
- le gestionnaire d'action et sa liste d'images ;
- le `TClientDataset` et son `DataSource` ;
- le `TTimer` pour afficher l'heure dans la partie droite de la barre d'outils.

À l'exécution, si on clique sur le bouton de la JVCL pour changer la couleur de la première colonne, l'affichage ressemble à ce qui est montré figure 2 :

[1]. La JVCL est une bibliothèque freeware et open source très riche que bon nombre de développeurs Delphi connaissent.

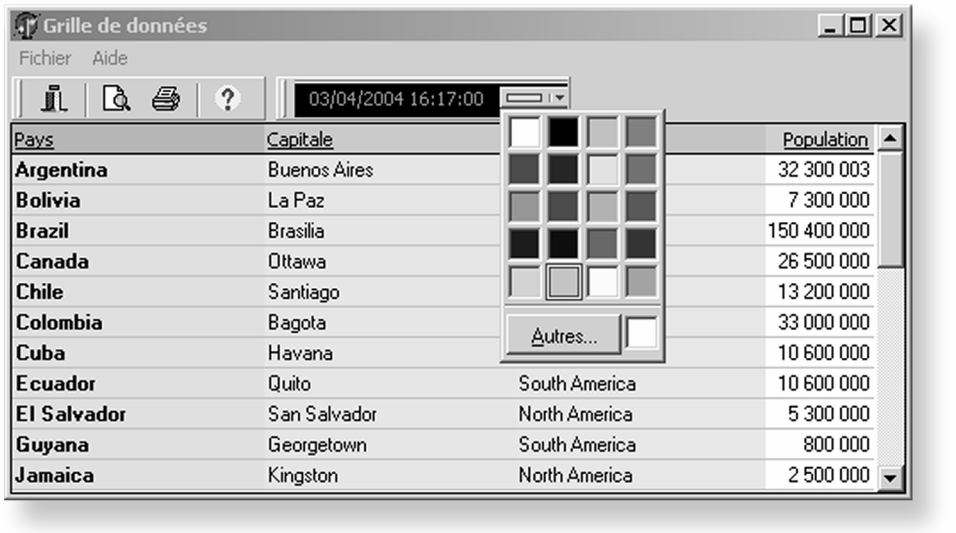
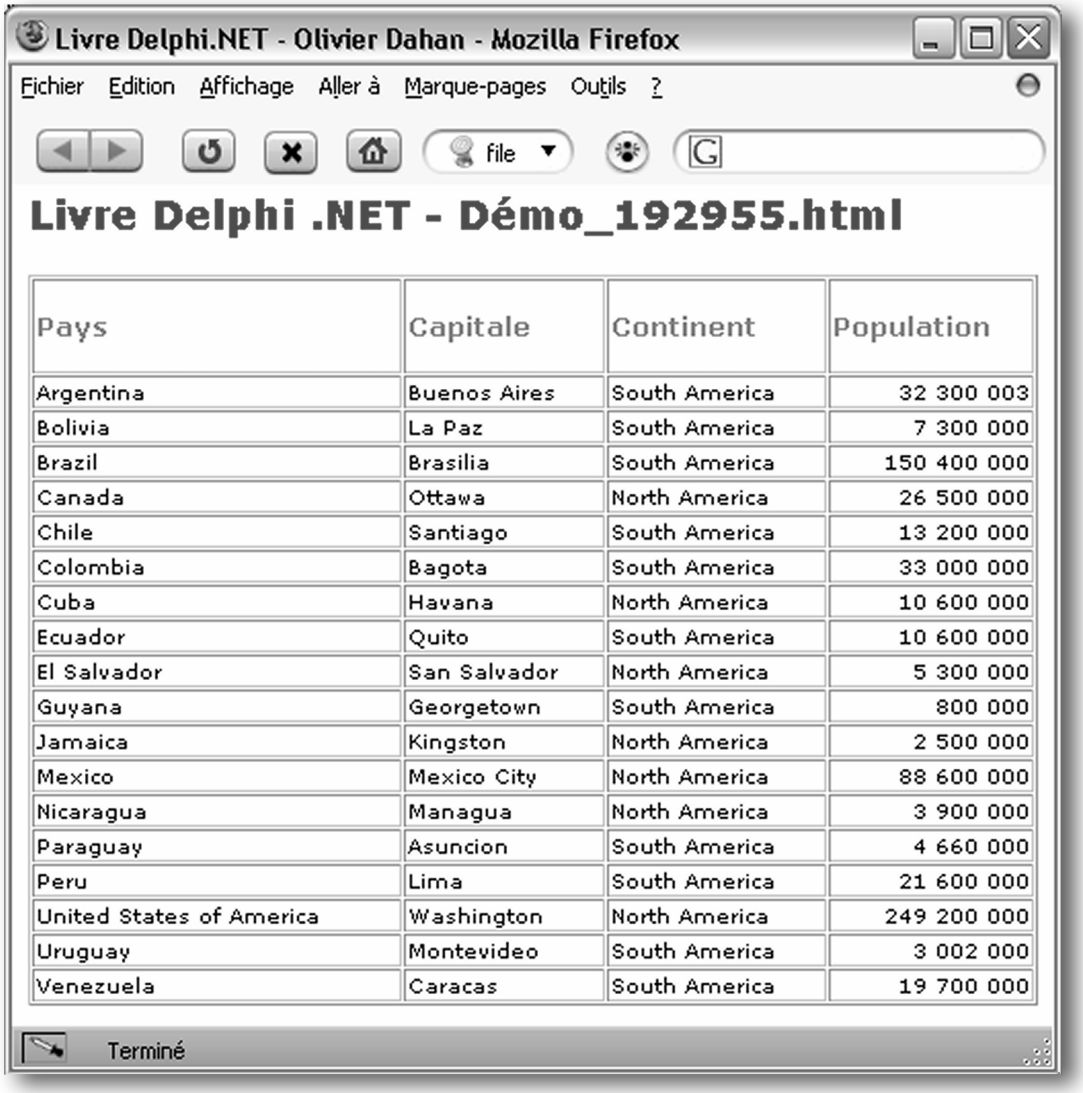


Figure 2

L'application Delphi 7 à migrer en cours d'exécution

Si nous cliquons sur le bouton de prévisualisation (identique à l'action par le menu Fichier/Prévisualisation), l'application génère un fichier HTML puis invoque le navigateur Internet installé sur la machine pour l'afficher, comme cela est montré figure 3 :



Pays	Capitale	Continent	Population
Argentina	Buenos Aires	South America	32 300 003
Bolivia	La Paz	South America	7 300 000
Brazil	Brasilia	South America	150 400 000
Canada	Ottawa	North America	26 500 000
Chile	Santiago	South America	13 200 000
Colombia	Bagota	South America	33 000 000
Cuba	Havana	North America	10 600 000
Ecuador	Quito	South America	10 600 000
El Salvador	San Salvador	North America	5 300 000
Guyana	Georgetown	South America	800 000
Jamaica	Kingston	North America	2 500 000
Mexico	Mexico City	North America	88 600 000
Nicaragua	Managua	North America	3 900 000
Paraguay	Asuncion	South America	4 660 000
Peru	Lima	South America	21 600 000
United States of America	Washington	North America	249 200 000
Uruguay	Montevideo	South America	3 002 000
Venezuela	Caracas	South America	19 700 000

Figure 3

La prévisualisation du code HTML généré via le navigateur Internet

Migration étape 1 : contrôles sous Delphi 7

Comme nous l'avons expliqué, avant même de vouloir passer notre code sous Delphi pour .NET, la première chose à faire consiste à contrôler le code dans sa totalité sous Delphi 7.

Étape Zéro !

Et oui… nous ne sommes pas des informaticiens pour rien et nous ne comptons pas à partir de 1 comme le commun des mortels mais bien à partir de 0, c'est tellement plus snob !

En réalité, il s'agit bien là d'une étape essentielle, cruciale, indispensable, bref vous avez certainement compris : nous parlons des sauvegardes.

> Pour une telle migration de code, ne travaillez **jamais** sur le code original, mais faites-en une copie dans un nouveau répertoire et travaillez sur cette copie !

Il est en effet bien plus aisé de faire des modifications sur un code qui fonctionne et qu'il reste possible de tester plutôt que de vouloir modifier de nombreuses choses en aveugle sous Delphi .NET, jusqu'à ce que le code puisse être enfin compilé et testé.

Le contrôle va consister principalement :

1. à insérer des commutateurs d'avertissement dans chaque unité ;
2. à changer tout code suspect, notamment celui qui sera indiqué par les avertissements mis en route précédemment (point 1) ;
3. à passer les fiches en mode texte si ce n'est déjà fait (étape optionnelle) ;
4. à supprimer ou changer les composants non standard par des équivalences au plus proche dans la VCL .NET.

Étape 1.1 – Les commutateurs

Notre application contient trois fichiers de code, le projet lui-même (`.DPR`), le code de la fiche principale (`Test1.pas`) et le code de l'unité déclarant la classe d'impression HTML (`DBGridPrint.pas`). Nous allons ajouter les lignes suivantes en haut de chacune de ces unités :

```
{$WARN UNSAFE_TYPE ON}
{$WARN UNSAFE_CODE ON}
{$WARN UNSAFE_CAST ON}
```

Ensuite, nous allons vérifier les options du compilateur et supprimer celles qui poseront problème. À l'origine les options de compilation du projet étaient celles présentées figure 4.

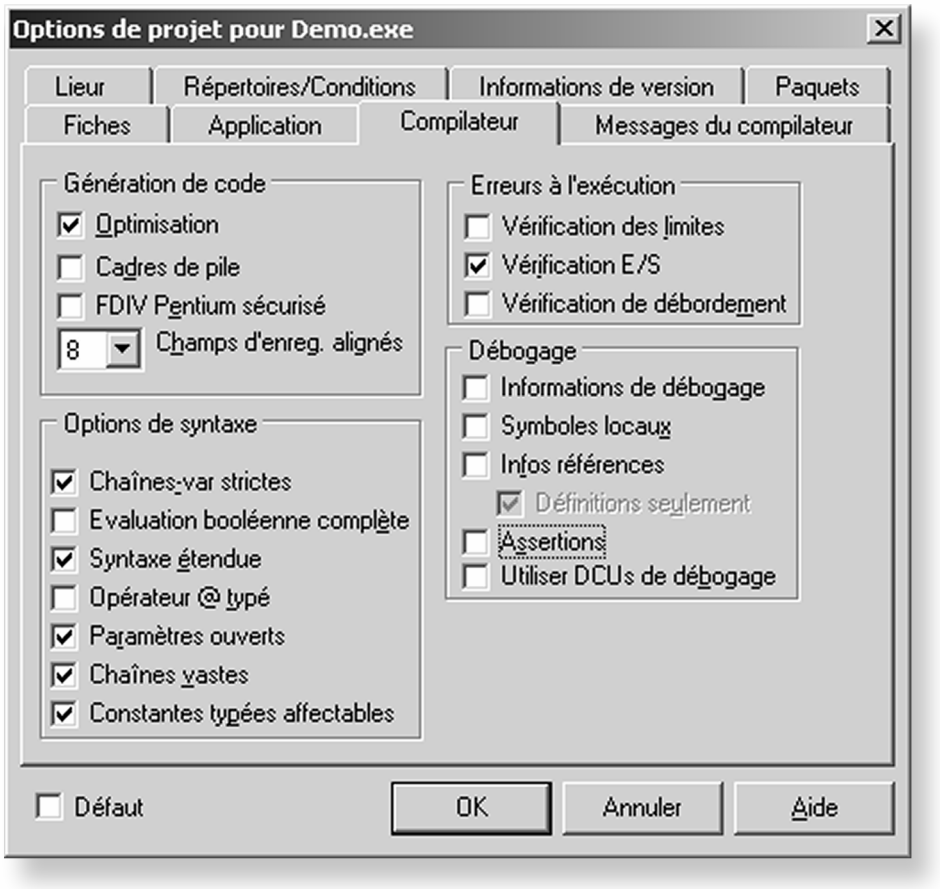


Figure 4

Les options des compilations originales

La figure 5 montre les changements que nous avons opérés.

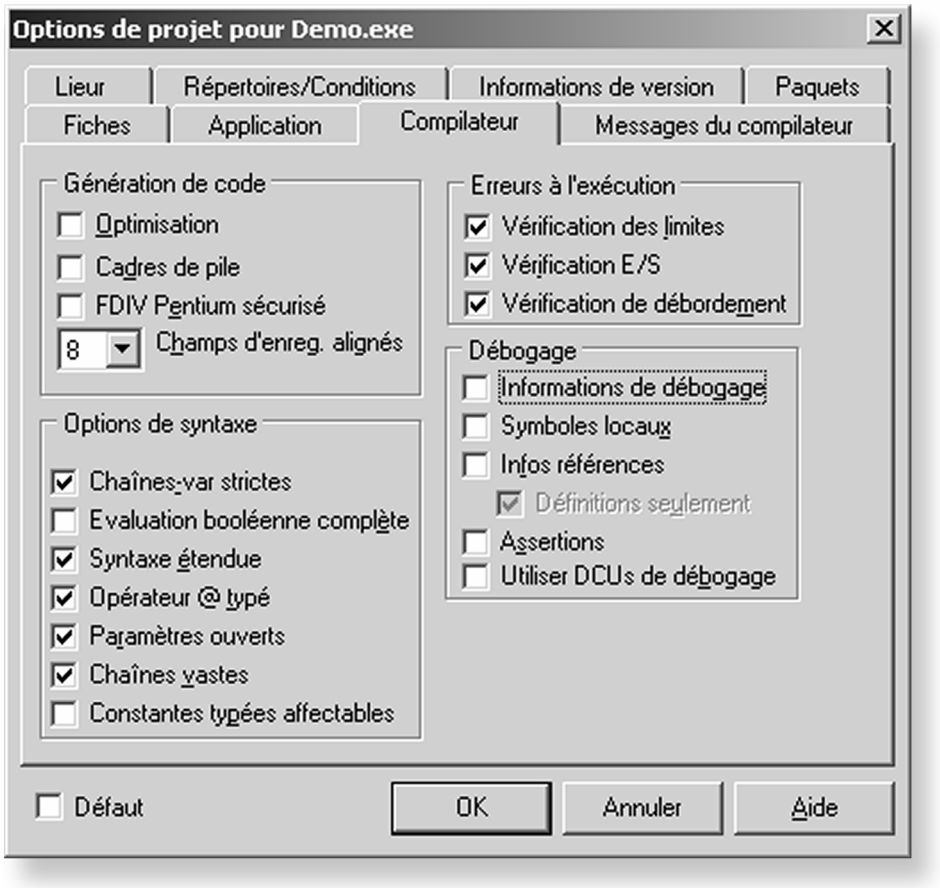


Figure 5

Les nouvelles options de compilation

Les options modifiées sont :

- La suppression de l'optimisation qui n'a plus d'intérêt et dont la présence peut modifier le comportement du code.
- Le typage fort de l'opérateur `@` a été ajouté pour détecter des opérations pointeur qui ne seront plus autorisées.
- Les constantes typées affectables ont été supprimées.
- Les vérifications des erreurs d'exécution ont été mises en fonction pour lever les bogues éventuels (car nous exécuterons notre application avant de la migrer totalement pour s'assurer de son bon fonctionnement, ce qui sera l'occasion de supprimer d'éventuels anciens bogues inconnus).
- Les options de déboguage ne sont pas mises en route pour l'instant, cela ne sera utile que si nous détectons des erreurs de fonctionnement lors des exécutions de test.

Étape 1.2 – La correction du code

Une fois les modifications des commutateurs effectuées, nous allons construire l'application. Il ne faut en effet pas se contenter d'une simple compilation ou d'une vérification de syntaxe. La première ne force pas la recompilation des unités déjà compilées et non modifiées et nous pourrions passer à côté d'un problème ; la seconde ne vérifie que la syntaxe et non la totalité des erreurs possibles. Pour tester convenablement le code, il faut lancer un *build* complet par le menu Projet. Le nôtre s'appelant tout simplement `Demo`, l'appel du menu sera Projet/Construire `Demo`.

La première erreur que nous rencontrons est la suivante :

```
[Erreur] DBGridPrint.pas(47) : La partie gauche n'est pas affectable
```

Cette erreur se produit sur la portion de code suivante :

```
implementation


uses
  ShellAPI, DB;


Const DummyConst : Integer = 12; // pour la démonstration


constructor TDBGridPrint.Create(AOwner: TComponent);
begin
  Inherited Create(AOwner);
  FHandle := 0;
  // code inutile pour forcer l'utilisation de la constante
  // aux fins de démo
  DummyConst := DummyConst + 1; // ERREUR ICI A LA COMPILATION
end;
```

Le code de l'unité contient en effet une constante initialisée qui est utilisée ensuite comme une variable. En supprimant l'option de compilation *Constantes typées affectables*, nous avons permis la détection de ce problème.

Le code sera corrigé en changeant la constante en une variable :

```
var DummyConst : Integer = 12; // const transformée en var
```

Cette modification est suffisante puisque les variables initialisées sont, elles, autorisées sous Delphi pour .NET.

Nous relançons ensuite le build. Delphi nous indique trois problèmes potentiels :

```
[Avertissement] DBGridPrint.pas(57): Type non protégé 'TBookmark'
[Avertissement] DBGridPrint.pas(98): Type non protégé 'SavePlace: Pointer'
[Avertissement] DBGridPrint.pas(135): Type non protégé 'SavePlace: Pointer'
```

Le premier avertissement concerne un type non protégé, c'est-à-dire un pointeur. En effet, `TBookmark` est un type déclaré comme

`Pointer` dans les sources de Delphi, ce que montre cet extrait du code de l'unité `DB.pas` de la VCL de Delphi 7 :

```
TBookmark = Pointer;
```

Toutefois, comme il s'agit d'un élément de la VCL qui a été portée sous .NET, vous n'avez pas à vous soucier de ce problème : la VCL .NET règle elle-même ce qui concerne son propre code. Nous n'effectuons donc aucune modification ici.

À titre indicatif, si vous regardez le code de la VCL .NET, vous verrez que le type `TBookmark` n'est plus un `Pointer` mais un `IntPtr`, un type spécial (descendant de `System.ValueType`, un type par valeur) de la plate-forme .NET déclaré dans l'espace de nom `System` du framework Microsoft. La VCL s'est donc adaptée et utilise un type parfaitement légal en place et lieu du pointeur original.

Afin de ne pas être brouillé dans la suite de notre migration par ce type non sécurisé qui, en fait, le devient puisqu'il appartient à la VCL et non à notre code propre, nous allons déconnecter le contrôle de type autour de la déclaration :

```
{$WARN UNSAFE_TYPE OFF}
  SavePlace : TBookmark;
{$WARN UNSAFE_TYPE ON}
```

Nous ne déconnectons que la détection des types non sécurisés (commutateur `UNSAFE_TYPE`) puisque l'avertissement concerne un type. Nous ne déconnectons rien d'autre, d'autres erreurs pourraient se cacher dans le code. Il est clair qu'ici nous n'aurons aucune autre surprise, mais cette démarche doit être systématique. Le meilleur moyen de rien oublier est d'appliquer la même logique instinctivement, mécaniquement.

Il reste encore deux avertissements à étudier. Ils concernent tous les deux un autre type non protégé, une variable déclarée sous la forme d'un pointeur. Le code concerné est le suivant :

```
SavePlace := DbGrid.DataSource.DataSet.GetBookmark;
```

et :

```
DbGrid.DataSource.DataSet.GotoBookmark(SavePlace);
```

Il s'agit encore du problème des `Tbookmark` déclarés sous la forme de pointeur sous Delphi 7. Mais il ne suffit pas ici de déconnecter les avertissements comme nous l'avons fait précédemment. En effet, si l'utilisation du pointeur dans la VCL est réglée par la VCL elle-même, c'est nous qui avons déclaré dans note code la variable que nous utilisons, à savoir `SavePlace`… Nous devons donc déconnecter les avertissements sur les deux lignes en question mais aussi adapter la déclaration de `SavePlace` !

Première action : déconnecter les avertissements inutiles dans le code aux deux endroits incriminés :

```
{$WARN UNSAFE_TYPE OFF}
SavePlace := DbGrid.DataSource.DataSet.GetBookmark;
{$WARN UNSAFE_TYPE ON}
// -------------
{$WARN UNSAFE_TYPE OFF}
DbGrid.DataSource.DataSet.GotoBookmark(SavePlace);
{$WARN UNSAFE_TYPE ON}
```

La seconde action consiste à modifier le code déclarant la variable `SavePlace`. En réalité, c'est le code sur lequel nous avons en premier désactivé les avertissements. Il n'y a rien de spécial à faire donc. Par acquit de conscience, nous devons tout de même vérifier que la variable est bien déclarée comme un `TBookmark`, type qui sera migré correctement par la VCL elle-même, et non comme un `Pointer` : cela marcherait sans problème sous Delphi 7 mais plus sous Delphi pour .NET. Comme la variable est déclarée convenablement, nous arrêtons là les investigations.

Les étapes 1.1 et 1.2 de la migration sont terminées, Delphi 7 ne nous indique plus aucune erreur. Passons à la suite.

… Mais pas si vite !

Il est en effet fort imprudent de continuer sans avoir réellement reconstruit une dernière fois le projet et sans l'avoir exécuté et testé totalement ! Tout doit fonctionner comme avant et seul un test exhaustif le confirmera. Cela est fastidieux, nous le concédons, mais seule la rigueur de cette approche vous garantit une migration sans bogue ni soucis. Perdre systématiquement un peu de temps dont la durée est connue est bien préférable que d'en perdre beaucoup ultérieurement sans être sûr de la durée de l'intervention (et encore, si vous trouvez le bogue avant que le logiciel ne soit installé chez des centaines d'utilisateurs…).

Notre code a été de nouveau testé, il fonctionne toujours comme le cahier des charges le précise, notre plan de test le confirme, nous pouvons maintenant, et seulement maintenant, passer à la suite.

Plan de test

Les tests d'un logiciel ne doivent pas être aléatoires ni dépendre du testeur et encore moins du temps dont on dispose. Ils doivent faire l'objet de plans précis et écrits. Ces plans de test sont idéalement conçus en même temps que le logiciel par une équipe spécialisée indépendante de celle du développement et sur la seule base du cahier des charges, module par module, fonction par fonction. Des logiciels permettant d'automatiser les plans de test existent, BDS est même fourni avec DUnit que nous vous conseillons d'utiliser dans toutes vos réalisations.

Étape 1.3 – Passer les fiches en mode texte

Cette étape n'est pas obligatoire, mais il est plus pratique que les DFM soient en texte, pour chercher par exemple toutes les occurrences d'un type de composant par l'explorateur de Windows.

Cela peut sembler trivial mais ici aussi il faut agir avec rigueur. Pour être certain de n'oublier aucune fiche du projet dans notre vérification, nous allons utiliser la console de Windows pour créer un listing de tous les DFM de notre projet.

Pour cela nous ouvrons une console par le menu Démarrer de Windows puis Exécuter. Nous donnons le nom de la console (CMD sous Windows 2000/XP et équivalent) et nous validons. La console s'ouvre. Nous changeons de répertoire pour nous placer dans la racine du projet en cours de migration. La commande suivante est enfin exécutée :

```
DIR *.DFM /S >LESDFM.TXT
```

Cette commande va créer un listing, appelé `LESDMF.TXT` dans la racine du projet, qui contiendra tous les fichiers DFM du répertoire racine et de tous les sous-répertoires éventuels du projet (option `/S` du `DIR`).

Le mieux est d'imprimer ensuite ce fichier texte et durant le contrôle (et l'éventuelle transformation) de chaque fiche, de cocher chaque ligne en fonction de la progression. Voici à quoi ressemble le listing de notre petit projet ne contenant qu'une fiche :

```
Le volume dans le lecteur D s'appelle D
Le numéro de série du volume est 8CC4-665B
Répertoire de D:\_D8BOOKOD\CHAPS\07\Code\Portage\Appli\D7\Migration
04/06/2004  16:15               38 350 Test1.dfm
               1 fichier(s)           38 350 octets

     Total des fichiers listés :
               1 fichier(s)           38 350 octets
               0 Rép(s)   6 714 863 616 octets libres
```

Armé de votre liste, il suffit de balayer le projet sous Delphi 7, de charger chaque fiche et de vérifier par un clic-droit sur chacune d'elles que l'option « DFM Texte » est bien cochée. Les modifications sont enregistrées, la fiche fermée avant de passer à la suivante.

Étape 1.4 – Les composants non standard

Nous abordons ici l'étape la plus critique. Il est difficile de vous fournir un plan de marche systématique car il existe autant de problèmes différents que de composants non standard ! Il est bien évident que si vous utilisez un bouton légèrement amélioré, il sera assez facile de se passer des fioritures et d'utiliser un bouton standard. Par contre, si le composant non standard assure une

fonction vitale du logiciel, le problème peut être insoluble, sauf à attendre que l'éditeur le propose pour .NET ou que vous puissiez le migrer vous-même. Nous étudierons cette dernière possibilité dans la partie 2 (article suivant) ; il faut pour cela que vous possédiez les sources du composant en question (règle d'or : ne jamais utiliser un composant fourni sans ses sources).

Lorsque le composant peut être remplacé par un équivalent de la VCL, les choses peuvent se dérouler de deux façons :

- Le faire à la main, ce que nous allons étudier.

- Utiliser l'option « Replace Components » des GExperts, une série d'experts gratuits pour l'IDE pour Delphi Win32 qu'on trouve facilement sur Internet (Google vous dira tout !).

La seconde méthode est plus simple puisqu'il suffit de cliquer sur le composant à changer puis d'appeler l'expert et de lui fournir la classe d'échange. Tout le reste est automatique. Il faut par contre faire attention aux événements et autres propriétés par adresse qui ne sont pas repris et s'assurer que le nouveau composant simule réellement bien ce que faisait l'autre.

Nous allons ici voir la méthode manuelle grâce notre application exemple. Celle-ci utilise effectivement un composant non standard, le `TJvColorButton`, dont on voit l'effet en cours d'utilisation figure 2. C'est un bouton qui, lorsqu'on clique dessus, ouvre un dialogue permettant de choisir une couleur.

Dans l'ordre des priorités, et pour chaque composant non standard, nous regarderons :

1. Si un composant très proche existe dans la VCL .NET. Si c'est le cas, nous le préférerons à toute autre solution.

2. Si un composant très proche existe dans le framework .NET Microsoft. Dans ce cas, il sera possible de l'intégrer en le transformant en composant VCL (nous expliquons la technique dans l'article partie 2).

3. Si le composant non standard peut être migré sous .NET. Le cas échéant, il suffit de lui appliquer les mêmes règles de migration qu'à tout autre code. Le succès d'une telle opération dépend énormément du code du composant, de sa qualité, de sa technicité, et des relations qu'il entretient avec les API Win32.

Concernant le composant utilisé dans notre application de démonstration, nous allons opter pour la version la plus simple : prendre un bouton standard de la VCL. En fait nous choisirons plutôt un `TSpeedButton`, mieux adapté à la situation. Pour le dialogue des couleurs, nous nous satisferons de l'ouverture d'un

dialogue classique lorsque l'utilisateur clic sur le bouton, ce qui est tout à fait acceptable pour un portage. Libre à chacun ensuite, selon le temps dont il dispose, d'affiner les applications portées. L'essentiel étant de préserver les fonctionnalités. Le *look & feel*, pour important qu'il soit, n'est pas prioritaire dans la majorité des applications.

L'échange du composant

La méthode la plus directe consisterait à supprimer purement et simplement le composant puis à poser le nouveau à la place. C'est envisageable dans certains cas mais le plus souvent, cela créera beaucoup de problèmes. Imaginons en effet une version spéciale d'un `TPanel` ou de tout autre conteneur… Si nous le supprimons de cette façon, nous perdons en même temps tout ce qu'il contient. Autant réécrire toute l'application dans un tel cas !

La solution consiste plutôt à modifier manuellement à la fois le fichier source .PAS et le .DFM.

> **Attention, danger !**
> La méthode que nous allons exposer ici est potentiellement risquée, nous vous conseillons vivement d'effectuer des sauvegardes du projet avant de vous lancer ainsi qu'après chaque grande session de modifications couronnée de succès.

Maintenant que nous savons par quelle classe nous allons remplacer le bouton non standard, passons à la modification de la fiche.

En premier lieu, nous allons noter les propriétés exactes du composant original. Pour ce faire, nous cliquons sur le composant et nous le copions ensuite dans le presse-papiers par Ctrl-C. Après ouverture du bloc-notes de Windows, nous collons le contenu du presse-papiers. Dans notre cas, nous obtenons le texte suivant :

```
object JvColorButton1: TJvColorButton
  Left = 121
  Top = 2
  Height = 13
  OtherCaption = '&Autres...'
  Options = [cdAnyColor]
  Color = 16768477
  OnChange = JvColorButton1Change
End
```

Cela nous permet de garder une trace de toutes les propriétés qui ont été modifiées et surtout d'obtenir la liste de tous les événements gérés.

Plaçons momentanément un composant de la nouvelle classe sur la fiche (ici un `TSpeedButton` donc). Nous sauvegardons l'unité, ce qui force l'EDI de Delphi à ajouter dans les `Uses` le nom de

l'unité du composant s'il n'y est pas déjà. Ceci est essentiel pour le reste de la manœuvre. Nous coupons maintenant le composant par Ctrl-X pour le coller à la suite dans le bloc-notes. Nous allons utiliser un `TSpeedButton`, ce qui donne le texte suivant :

```
object SpeedButton1: TSpeedButton
  Left = 296
  Top = 2
  Width = 23
  Height = 22
End
```

Nous allons maintenant, dans le bloc-notes, façonner en quelque sorte ce nouveau code pour qu'il mime au mieux le composant original. Observons d'abord les propriétés identiques comme la taille, la position et copions celles du composant original dans son successeur. Le code de ce dernier devient alors :

```
object SpeedButton1: TSpeedButton
  Left = 121
  Top = 2
  Width = 23
  Height = 13
End
```

Nous regardons maintenant les autres propriétés et tentons de les simuler. La propriété `OtherCaption` ne peut être reproduite facilement, elle s'affichait dans le menu déroulant du bouton pour ouvrir justement le dialogue des couleurs que nous allons utiliser directement. Les `Options` ne sont pas non plus reproductibles à première vue. En réalité, il s'agit des options du dialogue standard de sélection de couleurs qui seront ouvertes optionnellement par le composant. Ces options seront donc à placer manuellement dans le composant dialogue que nous ajouterons à la fiche. La propriété `Color` n'est hélas ici pas transposable puisque le `TSpeedButton` n'a pas de propriété équivalente. Nous envisagerons plus tard d'améliorer l'interface lorsque le logiciel tournera sous .NET. Reste l'événement `OnChange`. Que fait-il ?. Repassons la fiche en mode visuel (clic-droit sur le texte, option Voir comme fiche). Sélectionnons le composant et double-cliquons sur l'événement pour voir le code qui est :

```
procedure TForm1.JvColorButton1Change(Sender: TObject);
begin
 DBGrid1.Columns[0].Color := JvColorButton1.Color;
end;
```

Ce code modifie la couleur de la première colonne de la grille en reprenant celle qui a été choisie. Notre `TSpeedButton` ne permet pas de faire cela directement. Il faut un clic sur ce dernier pour ouvrir un dialogue de sélection de couleurs et affecter la couleur choisie à la colonne de la grille. Donc, le code actuel est un morceau de ce qui va devenir notre nouveau code. Nous allons

ainsi couper la partie utile (entre le `Begin` et le `End`) et la coller dans le bloc-notes. Une sauvegarde de l'unité supprimera automatiquement l'ancien gestionnaire (puisqu'il n'y a plus rien entre le `Begin` et le `End`).

Nous allons passer à la partie délicate de l'échange. Repassons la fiche en mode texte et cherchons le composant à modifier. Voici un extrait de ce que nous pouvons voir :

```
  ...
        Font.Style = []
        ParentFont = False
        Transparent = True
      end
      object JvColorButton1: TJvColorButton
        Left = 121
        Top = 2
        Height = 13
        OtherCaption = '&Autres...'
        Options = [cdAnyColor]
        Color = 16768477
      end
    end
  end
  object DataSource1: TDataSource
    DataSet = ClientDataSet1
  ...
```

Terminons la modification dans le bloc-notes du code du nouveau composant en lui donnant le même nom que l'actuel, ce qui donne, au final :

```
object JvColorButton1: TSpeedButton
  Left = 121
  Top = 2
  Width = 23
  Height = 13
End
```

Copions ce code dans le presse-papiers. Retournons sous Delphi et sélectionnons tout le bloc du composant actuel puis tapons Ctrl + V pour coller le nouveau composant à sa place. Le code de la fiche devient alors :

```
  ...
        Font.Style = []
        ParentFont = False
        Transparent = True
      end
object JvColorButton1: TSpeedButton
  Left = 121
  Top = 2
  Width = 23
  Height = 13
End
```

```
    end
  end
  object DataSource1: TDataSource
    DataSet = ClientDataSet1
  ...
```

L'indentation n'a aucune importance, nous avons d'ailleurs fait exprès que le code soit décalé pour que vous puissiez mieux visualiser la partie échangée.

Il ne reste plus qu'à repasser en mode fiche pour voir le résultat. Si tout va bien, le nouveau composant remplace l'ancien, ce qui est bien heureusement notre cas. Il reste toutefois deux choses importantes à faire :

1. modifier le type du composant dans la déclaration du `TForm` ;
2. supprimer des `Uses` l'unité de l'ancien composant.

La ligne de `TForm1` suivante :

```
JvColorButton1: TJvColorButton;
```

devient alors

```
JvColorButton1: TSpeedButton;
```

Après avoir simuler l'aspect, il nous reste à simuler le fonctionnement. Nous ajoutons ainsi un gestionnaire pour l'événement `OnClick` du bouton dans lequel nous ouvrirons un dialogue de sélection de couleurs (composant que nous ajoutons à la fiche). Une fois ce code de simulation saisi, il ne faut pas oublier d'ajouter le code supprimé que nous avons copié dans le bloc-notes ce qui donne au final le code du gestionnaire `OnClick` :

```
procedure TForm1.JvColorButton1Click(Sender: TObject);
begin
 ColorDialog1.Color := DBGrid1.Columns[0].Color;
 if ColorDialog1.Execute then
  DBGrid1.Columns[0].Color := ColorDialog1.Color;
end;
```

La première ligne de ce code simule le comportement de l'ancien composant en ouvrant le dialogue de sélection de couleurs ; la seconde ligne reproduit le comportement de l'ancien code. Aspect et comportement sont maintenant simulés, ce qu'une exécution de test nous confirmera.

Conclusion

Lors de cette première étape de migration, nous avons travaillé sous Delphi 7 uniquement pour contrôler et adapter le code original

autant que la partie visuelle. Après une nouvelle sauvegarde, notre code est près pour le grand saut : le chargement sous Delphi pour .NET !

C'est ce que nous verrons dans l'article suivant qui complète l'étude de la migration d'une application et qui présente ensuite la technique à appliquer pour migrer un composant VCL et la façon d'importer un contrôle Windows Forms en VCL.NET