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

© Copyright 2005 Olivier DAHAN
Reproduction, utilisation et diffusion interdites sans l'autorisation de l'auteur. Pour plus d'information contacter odahan@e-naxos.com

La plate-forme .NET

La plate-forme .NET est un ensemble formé d'outils (présents dans le SDK), de modules actifs au runtime (le CLR par exemple) et de classes formant une API très étendue pour les applications de bureau comme pour les applications Internet. Des langages exploitent cette nouvelle base comme C# ou Delphi.NET[1]. Du point de vue du développeur la plate-forme .NET peut être vue comme le successeur objet des API Win32. Mais cela va plus loin. Entrons dans le vif du sujet…

[1] Ces aspects sont traités dans les derniers ouvrages de Olivier Dahan parus chez Eyrolles

.NET : une vision du futur ?

Un framework est mot à mot un « cadre de travail ». Lorsqu'on parle du framework .NET, il s'agit donc de l'ensemble des outils et spécifications créant *ipso facto* un tel cadre dans lequel s'inscriront à la fois l'analyse et la réalisation d'une application. On parle aussi de plate-forme .NET comme on parlait de la plate-forme Windows, voire de la plate-forme Intel. À y regarder de plus près, .NET apparaît comme une version enfin achevée et objet des API Win32. Mais .NET est plus que cela, c'est aussi une vision de l'informatique de demain.

En effet, .NET est une réorganisation en profondeur dessinant l'informatique du futur et s'attaquant aussi bien aux langages qu'aux bibliothèques produites et aux rapports qu'elles entretiennent avec les plates-formes techniques sous-jacentes et dont le champ d'application s'étend de l'informatique domestique à la télévision interactive en passant par les PDA et l'informatique industrielle.

En bref, il s'agit aujourd'hui pour l'informatique d'être, selon le terme repris par Microsoft, « agile ». Agile donc souple, simplifiée, passe-partout et surtout intégrée. Intégrée à quoi ? À tout ! Un PDA doit pouvoir servir à passer une commande via une interface de type Web en surfant sur un site dont la base de données est utilisée simultanément par la gestion commerciale et comptable de la société et vous renvoyant une confirmation de commande sur votre téléviseur… À moins que ce ne soit votre frigidaire qui aura passé commande tout seul des denrées indispensables dont le stock sera descendu sous le minimum que vous aurez fixé et les produits livrés par un coursier averti sur son portable par un SMS transmis par l'ordinateur du supermarché local qui aura reçu votre commande et l'aura préparée…

Science-fiction ? Vision « microsoftienne » mercantile d'un avenir radieux pour ses actionnaires où même un grille-pain devra disposer d'une licence Windows en règle ? Vision d'horreur d'un futur orwellien ?

Tout ceci n'est pas exclu… les consommateurs que nous sommes étant très friands de technologie et s'offrant si docilement aux modes et aux lavages de cerveaux *staracadémiesques* dont on nous abreuve pour mieux nous diriger... La fascination en une science toute puissante, secrètement vécue comme un moyen de réparer nos errances désastreuses pour notre propre écosystème n'y ait pas pour rien non plus… Et puis l'humain modifie lui-même son futur, par son imagination débordante. Les communicateurs de Star Trek avec lesquels le capitaine Kirk pouvait, depuis une planète exotique, dialoguer avec l'Enterprise étaient-ils l'invention d'un scénariste visionnaire préfigurant nos téléphones portables

modernes ou bien est-ce l'idée du scénariste qui s'est lentement insinuée dans la société et a suscité à quelques ingénieurs le concept de nos portables ? Il y a fort à parier qu'il s'agisse en réalité d'une interaction à double sens et que nos fantasmes technologiques, qu'ils soient l'œuvre de scénaristes de science-fiction ou de directeurs marketing, influencent directement ce que l'industrie créera en réponse à la demande du public… Le peuple victime de ses propres rêves ? Qui peut dire exactement la part inconsciente du plaisir de tenir un portable de 60 grammes à la main que l'on doit aux communicateurs de Star Trek ? Qui sait, demain, la part qu'aura jouée la vision de Microsoft dans ce que nous considérerons alors comme indispensable ? L'idée de Microsoft est lancée et elle va inévitablement changer les choses par le simple fait qu'elle a été émise. Et que l'intention première soit une vision géniale du futur ou un simple pari commercial n'y change rien, l'idée va se propager jusqu'à modifier notre façon de percevoir l'informatique. Le marketing consiste à susciter l'envie chez le client et, dans ce domaine, la firme de Redmond a toujours été considérée comme imbattable, même par ses détracteurs. Ne boudons pas notre plaisir, ne gâchons pas de possibles évolutions bénéfiques par un passéisme illusoirement protecteur, mais restons vigilants…

L'architecture .NET

À l'heure où cet article est écrit, le framework est désormais disponible en version 2.0. La version 1.1 couvrait de très nombreux domaines, des applications de gestion classiques aux services Web en passant par les sites services Windows et les applications console ce qui en faisait une base solide pour réellement développer sous .NET. Elle restera certainement encore utilisée quelques temps mais les améliorations de la version 2.0 feront certainement pencher assez vite la balance en la faveur de cette dernière.

Le point fort de .NET est d'intégrer des technologies existantes dans un tout cohérent. Par exemple, tout ce qui concerne les communications de données reste basé sur SOAP, XML et HTTP. En aucun cas Microsoft n'a cherché à bouleverser les acquis, ce qui est une nouveauté.

La seule différence technique majeure est l'adoption d'un système d'exécution virtuel (VES) et qui se trouve désormais au cœur de toute la structure .NET et sur laquelle tout repose.

L'ensemble .NET peut être ainsi divisé en :

- Un framework proprement dit (bibliothèques de classes) ;

- des outils de développement comme Visual Studio .NET ou Web Matrix (EDI gratuit pour ASP.NET);
- un ensemble de serveurs comme MS SQL Server ou BizTalk Server ;
- des logiciels fronteaux comme Windows XP, Windows CE, Office XP.

On peut aussi préférer un autre angle de vue et découper .NET en quatre pôles :

- les outils de développement et les librairies (langages C#, VB.NET, J#, Visual Studio .NET, des composants de développement, la librairie commune) ;
- les techniques Web incluant les services Web et les sites dynamiques avec ASP.NET ;
- la gestion des données avec ADO.NET et les serveurs Microsoft spécialisés pour le B2B, les e-mails, le stockage de données ;
- le support de plates-formes techniques telles que les téléphones cellulaires, les PDA, les consoles de jeux.

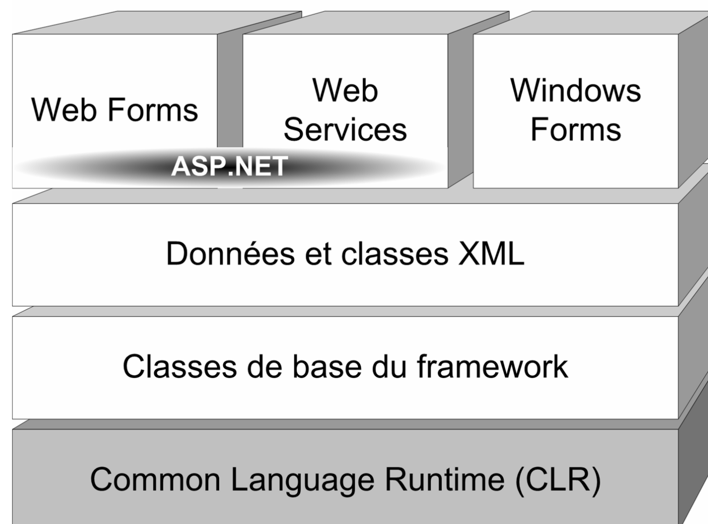

Figure 2.1

Le framework .NET

La figure 2.1 montre les différentes couches du framework. Au socle de l'édifice se trouve le CLR, Common Language Runtime[2],

[2] Runtime du langage commun (machine virtuelle)

c'est-à-dire la machine virtuelle de .NET, celle qui exécute le code IL appelé aussi CIL (Common Intermediate Language[3]), le pseudo-code du CLR.

Machine virtuelle ou non ?

Si l'on regarde de plus près le fonctionnement de .NET, il s'avère que le terme de « machine virtuelle » tel qu'on l'entend par exemple sous Java, c'est-à-dire un interpréteur de pseudo-code, est mal choisi , Microsoft semble d'ailleurs refuser ce terme. En effet, le pseudo-code .NET n'est jamais interprété comme cela se fait sous Java mais compilé avant d'être exécuté. Bien que la machine virtuelle .NET prenne en compte des tâches similaires à celles de Java (gestion de codes intermédiaires, gestion mémoire, ramasse-miettes et sécurité), il existe une nuance importante. Le terme « machine virtuelle » employé à propos du CLR de .NET doit donc être pris uniquement comme une comparaison simplificatrice.

Au-dessus du CLR se trouve l'ensemble des librairies de classes du framework, couche sur laquelle repose la gestion des données. Au sommet de cette construction, se trouvent deux blocs, celui des Windows Forms (WinForms), ensemble de classes permettant la conception d'IHM[4] Windows et celui appelé ASP.NET. Ce dernier n'a pas grand-chose à voir techniquement avec ce qu'on désigne par ASP sous Win32. ASP.NET est un ensemble de classes permettant à la fois la conception de sites dynamiques, la création d'IHM pour le Web, les WebForms et la conception de services Web.

Les bases de .NET

L'ensemble .NET repose sur le CLR, le runtime en charge de l'exécution du code CIL.

Le CLR

Ce runtime est le moteur de .NET, c'est lui qui est en charge de l'exécution des logiciels écrits pour .NET compilés en CIL. Il met en œuvre un contrôle strict des types et du code, le CTS (Common Type System) qui assure que le code géré (*managed*) est auto-descriptif et permet l'interaction parfaite entre différents codes conçus pour .NET. Le CLR gère :

- la compilation en code natif et l'exécution du code CIL ;
- la gestion de la sécurité ;

[3] Langage Intermédiaire Commun

[4] IHM : Interface Homme-Machine

- la gestion de la mémoire ;
- la gestion des processus ;
- la gestion des *threads* (tâches).

Les métadonnées contenues dans les fichiers exécutables .NET informent le CLR sur les éléments suivants :

- l'identité du code ;
- la version, la culture[5] et l'éditeur ;
- les déclarations de types pour le code interne et exporté (classes, interfaces, méthodes, champs, événements…) ;
- la référence vers les types utilisés ainsi que la version de ces références ;
- les attributs personnalisés utilisés.

Grâce à l'ensemble de ces informations et à ses propres caractéristiques, le CLR est en mesure d'offrir aux applications *managées*[6] une meilleure gestion des exceptions, la possibilité d'utiliser des composants écrits dans d'autres langages, une meilleure gestion de la sécurité des applications et une meilleure gestion des ressources, mémoire comprise. La figure 2.2 montre comment les IDE et les langages se positionnent par rapport au framework et au CLR.

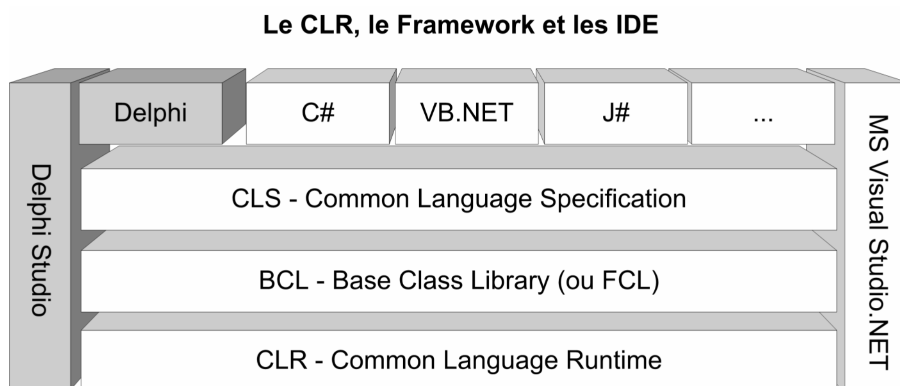

Figure 2.2

Le CLR, le Framework et les IDE

---

[5] le terme « culture » sous .NET désigne l'ensemble des informations de localisation de l'utilisateur (langue et paramètres régionaux principalement).

[6] « managé » est un anglicisme venant de « managed » voulant dire « géré ». Le « code managé » est celui qui est exécuté sous contrôle du CLR .NET. Certains préfèrent dire « code géré » plus français mais un peu vague.

La BCL ou FCL

Se plaçant directement au-dessus du CLR, la BCL ou Base Class Library (bibliothèque des classes de base) offre de très nombreuses classes, interfaces et types qui forment le socle des développements sous .NET. La BCL est aussi appelée FCL pour Framework Class Library (bibliothèque de classes du framework). La BCL permet d'unifier les développements puisqu'elle fournit l'ensemble des outils de base aux applications. En ce sens, on peut la comparer aux API de Windows mais elle est bien plus élaborée, et surtout développée en suivant le paradigme objet là où les API ne sont qu'une suite de procédures et fonctions éparses sans véritable unité.

La BCL et CLR forment à eux deux l'essentiel du framework et offre aux applications managées un support particulièrement performant pour les exceptions, les entrées/sorties, le mécanisme de réflexion (équivalent de l'introspection Java et des RTTI Delphi), la gestion de la sécurité, l'accès aux données, la gestion des IHM dites « riches » pour Windows et le Web, et bien d'autres services.

Les espaces de nom

Sous .NET toute classe est repérée dans un espace de nom[7] (*namespace*). Ce procédé simple et puissant permet d'éviter les conflits de noms entre différentes bibliothèques.

La gestion de ces espaces est particulièrement intuitive puisque chaque niveau de sous-espace est séparé des niveaux supérieurs par un simple point dans le nom. Ainsi, on trouve le framework de nombreux espaces de noms, chacun étant spécialisé dans un type de service. Ce procédé, en dehors de prévenir les conflits de noms, apporte une hiérarchisation des bibliothèques. Un bon exemple est le framework lui-même et ces milliers de classes hiérarchisées grâce à ce procédé.

CIL, CLI, JITer et code managé

Le CIL est le pseudo-code généré par les compilateurs des langages compatibles .NET. Ces langages sont d'ores et déjà nombreux et à côté de ceux produits par Microsoft tels que C# (créé par le concepteur du langage de Delphi Anders Hejlsberg), J# (un Java pour .NET), VB.NET (version .NET de Visual Basic) et Managed

[7] On trouve aussi l'expression « espace de nommage » pour désigner ce procédé. Ce néologisme utilisé dans notre précédent ouvrage ne nous semblant pas élégant nous lui préfèreront ici « espace de nom ».

C++ (une version spéciale de C++ pour .NET), nous trouvons Delphi .NET et son Pascal Objet mais aussi : COBOL.NET, Fortran.NET, Ruby.NET, P# (un Prolog pour .NET), S# (Smalltalk pour .NET), PHP.NET…

Une vingtaine de langages existaient pour .NET à sa sortie, il faut vraisemblablement en compter plus du double aujourd'hui.

CIL vs MSIL ou l'universalité de .NET

Le CIL, le langage commun intermédiaire, est aussi appelé MSIL pour Microsoft Intermediate Language lorsqu'il est adapté à l'implémentation spécifique Microsoft du framework .NET. En effet, .NET peut exister pour d'autres platesformes que Windows et être mis en œuvre par d'autres éditeurs. Il existe, par exemple, un framework .NET en cours de conception sous le nom de MONO. C'est un projet Open Source mené sous l'égide de Novell afin de créer une implémentation gratuite du framework. Les premières versions sont déjà distribuées pour Windows, mais également pour Linux avec Red Hat, SuSE, Debian, Mandrake…

L'infrastructure sur laquelle le CLR se repose s'appelle CLI, pour Common Language Infrastructure. CLI dispose d'une spécification publique et ouverte qui assure dès maintenant la propagation rapide de .NET (autant dans la diversité des langages existants et à venir que dans les diverses implémentations dont MONO).

Le CLI est ISO

Confirmant l'ouverture de .NET, il est important de noter que depuis avril 2003, un sous-ensemble du CLI a été porté au rang de standard international par l'ISO/IEC (International Organization for Standardization/International Electrotechnical Committee).

Lorsque le CLR exécute un programme compilé en CIL, il commence par vérifier si ce fichier a déjà été compilé en langage machine. Si tel est le cas, il ouvre l'image binaire et l'exécute ; sinon, il invoque le JITer (Just In Time Enhanced Runtime) qui a la charge de traduire le CIL en un véritable programme en langage machine pour la plate-forme hardware cible.

C'est là que réside la force de .NET : le CIL n'est pas exécuté par un interpréteur, lent par essence. Ce n'est qu'un langage pivot permettant la diffusion d'exécutables qui seront réellement compilés (au sens habituel) lors de la première utilisation. Cette astuce permet de diffuser le même exécutable .NET qui, potentiellement, fonctionnera sur différentes plates-formes de façon identique. Le JITer de .NET n'est pas en soi une révolution puisqu'il existe aussi des JITs pour Java. Mais à la différence de ce dernier qui a introduit le JIT pour pallier son principal défaut (le fait d'être interprété et lent), .NET intègre et étend le principe de compilation à la volée, classe par classe. L'avantage principal de .NET sur Win32 et Java est qu'il a été conçu ultérieurement et qu'il a su tiré leçon des erreurs de conceptions de ces derniers. Il répond à la pléthore des API non unifiées de Win32 par un framework cohérent, il répond au mélange interprétation / compilation de Java par l'intégration d'emblée d'un système de compilation efficace. Et

à défaut de promettre l'universalité trompeuse, car illusoire, de la portabilité de Java (de surcroît finalement inutile dans un monde dominé par la plate-forme PC), il offre d'emblée la pluralité des langages prenant ici une avance confortable sur ses compétiteurs limités à leur propre syntaxe.

Un code compilé et exécuté par le CLR est appelé un « code managé » (*managed code*) ou « code géré » si on préfère éviter le néologisme.

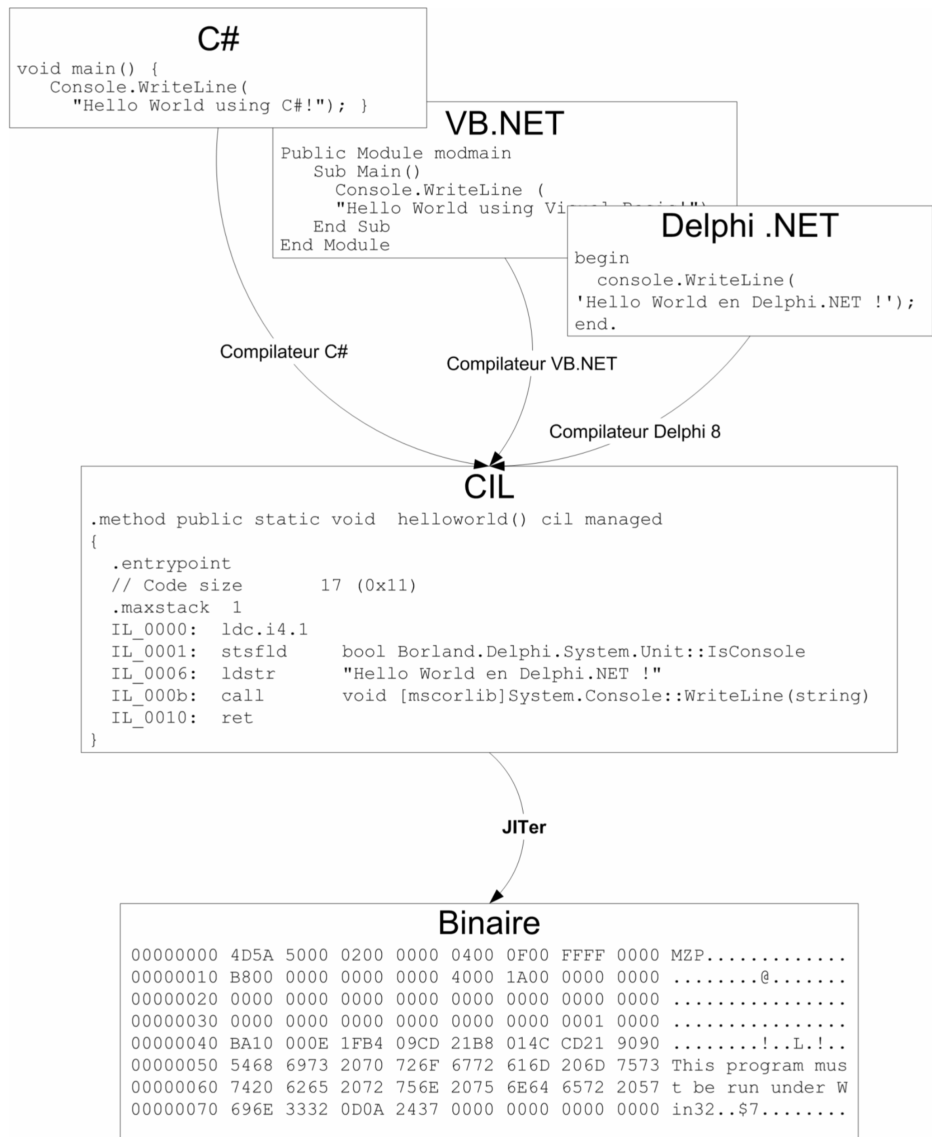

Figure 2.3

Du code source au binaire exécuté

La figure 2.3 présente la transformation d'un code source écrit en différents langages .NET (C#, VB.NET et Delphi .NET), d'abord en code CIL par le biais des compilateurs respectifs des langages considérés ici, puis en binaire par le JITer invoqué par le CLR.

Le célèbre « Hello World ! » passe ainsi par trois phases (code source, code CIL, code binaire) avant de pouvoir être exécuté sur une machine cible. Le code exécutable Delphi produit un fichier qui ne peut être exécuté que par .NET mais dont la reconnaissance est automatique sous Windows (l'installation du runtime ou du SDK .NET ayant ajouté un filtrage donnant la main au CLR lorsque l'exécutable est un fichier exécutable du framework).

Si on suit une logique futuriste, JITer, le compilateur qui traduit le CIL en binaire, pourrait être un jour intégré aux processeurs, devenant ainsi un langage machine à part entière. Mais pour l'instant Microsoft mise plutôt sur un mécanisme permettant justement à un code IL de s'exécuter sur des microprocesseurs différents. Toutefois, un processeur capable de comprendre le code IL directement permettrait d'atteindre certainement de meilleures performances, comme il existe des puces comprenant Java. Comme pour se dernier il y a fort à parier que si cela arrive ce le sera en premier pour les unités mobiles dans lesquels l'élévation des performances ne peut pas se faire aussi facilement que dans un PC en ajoutant un microprocesseur plus rapide et donc plus gourmand en énergie.

En attendant de tels processeurs dédiés CIL, le JITer effectue la compilation du langage intermédiaire en binaire. Il le fait à la volée, d'où son nom (Just In Time…) et cela classe par classe, selon trois modes :

- compilation de la classe invoquée et mise en cache ;
- compilation de tout le code et mise en cache ;
- compilation à la demande sans mise en cache.

Sous Windows c'est le premier mode qui est exploité par défaut alors que le troisième est préféré sur les équipements à faible mémoire, comme les assistants personnels. La seconde méthode permet d'accélérer une application dont la vitesse d'exécution est cruciale. On force alors la compilation totale de l'exécutable ainsi que son installation dans le cache (GAC, Global Assembly Cache ou Cache global des assemblages).

La Shadow Copy

Le code d'un assemblage .NET, EXE ou DLL peut être mis en cache en vue de son exécution, simplifiant les opérations de mise à jour. C'est la copie compilée binaire qui est alors utilisée et non l'original. Cette technique est appelée *Shadow Copy* que l'on traduirait en français par « copie miroir ». Cela permet de mettre à jour l'assemblage, même en cours d'exécution, ce qui simplifie les choses par rapport aux DLL et EXE Win32 qui ne peuvent pas être écrasés en cours d'exécution. La Shadow Copy se met en route volontairement par programmation (caractéristique de l'AppDomain) comme le fait ASP.NET pour faciliter la mise à jour des modules sur un site Web.

Les fichiers PE - Portable Executable

Sous Windows les fichiers exécutables (EXE et DLL) doivent se conformer au format PE (Portable Executable - exécutable portable). Ce format dérive de COFF (Common Object File Format) qui est une norme publique équivalente au format ELF de Linux.

Tous les fichiers PE commencent par un mini programme MS-DOS hérité des temps anciens où Windows n'était qu'une simple surcouche de DOS. Ce petit programme ne fait qu'afficher un message indiquant que Windows est nécessaire pour exécuter le fichier. Les deux premiers octets d'un programme DOS sont `0x5A4D`, c'est-à-dire les lettres `MZ`. La petite histoire retiendra qu'il s'agit des initiales de Mark Zbikowski, l'un des architectes de MS-DOS 2.0.

L'en-tête PE proprement dit suit l'en-tête DOS, tout cela précédant une série de sections (`.text`, `.data`, `.rdata`, `.rsrc`, ainsi que d'autres sections optionnelles).

.NET créé lui aussi des EXE mais ils sont de nature très différentes. Si les fichiers PE de DOS jusqu'aux versions les plus récentes de Win32 ne peuvent contenir que du code compilé en langage machine et des données, les exécutables .NET ajoutent des métadonnées et du code IL, indispensables au CLR. Les métadonnées permettent au CLR de savoir comment charger les classes et le code IL lui permet d'effectuer la compilation des différentes classes via le JITer. Pour satisfaire les besoins du CLR, Microsoft a été obligé d'étendre le format PE sans le remettre en cause. Pour ce faire, deux sections ont été ajoutées : un en-tête CLR et des données CLR. Ce nouveau format PE est illustré figure 2.4.

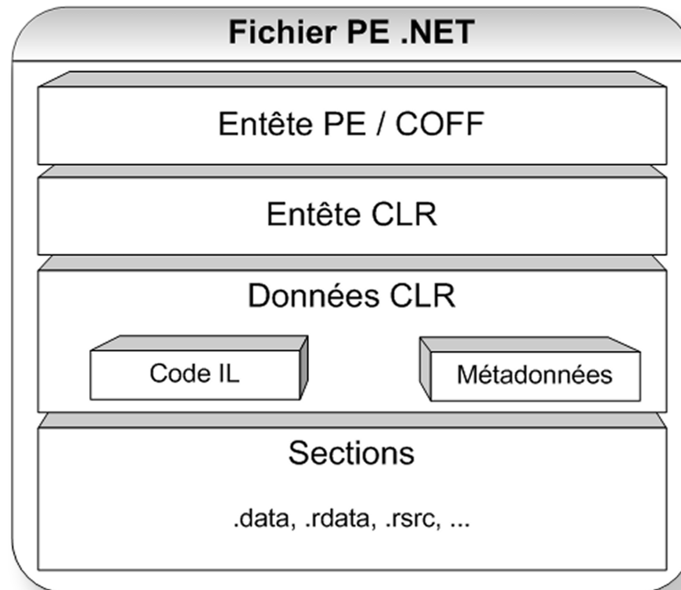

Figure 2.4

Le format de fichier PE de .NET

Les métadonnées et la réflexion

La structure d'un exécutable .NET lui permet d'exposer de nombreuses informations techniques sur l'application, notamment son code IL et ses métadonnées.

Ces dernières peuvent être visualisées facilement à l'aide de l'utilitaire ILDASM fourni avec le SDK du framework. Cet outil peut même désassembler proprement le code IL qui pourra être recompilé à l'aide de ILASM, le compilateur de CIL, comme un assembleur permet de compiler du code symbolique en code machine sous Win32 (ou d'autres plates-formes).

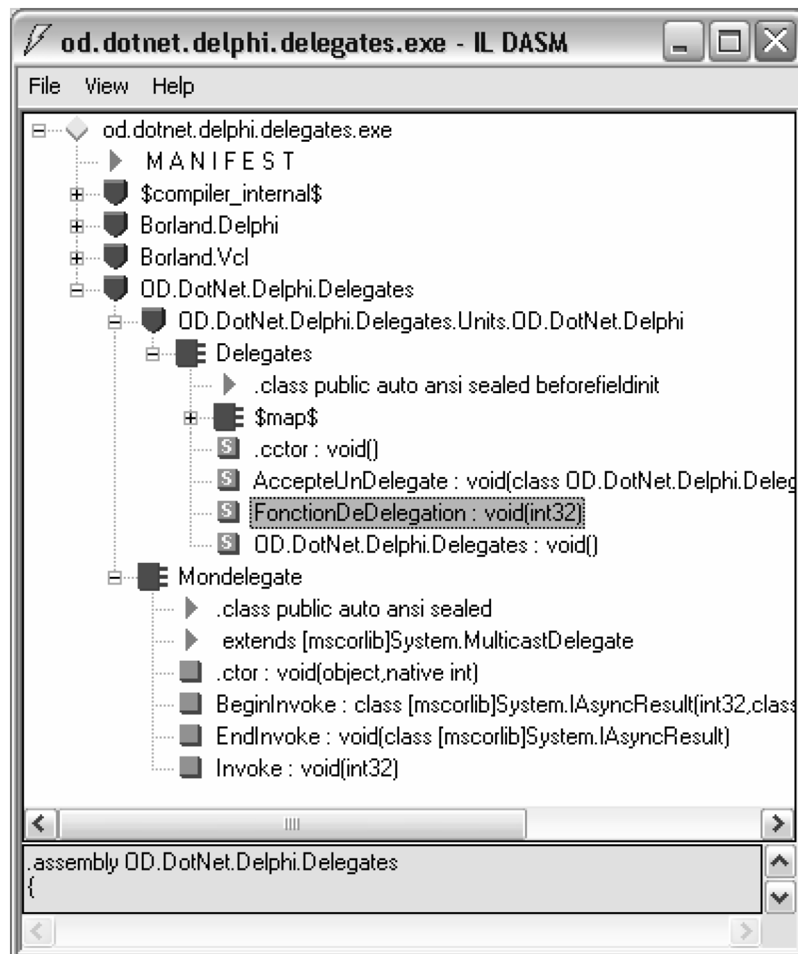

Figure 2.5

Les métadonnées vue par ILDASM

La figure 2.5 présente les informations exposées par le programme console `OD.DotNet.Delphi.Delegates.exe` compilé avec Delphi .NET. On peut voir les assemblages référencés ainsi que les méthodes publiées. On peut inspecter les unités, puis les classes et les méthodes contenues dans chaque classe. À partir de cette vue, il est aussi possible d'obtenir le désassemblage de l'application.

Ci-dessous la décompilation de la méthode principale du programme :

```
.method public static void
  FonctionDeDelegation(int32 i) cil managed
{
  // Code size       28 (0x1c)
  .maxstack  4
  .locals init ([0] object[] V_0)
  IL_0000:  ldstr      bytearray
  (4C 00 65 00 20 00 70 00 61 00 72 00 61 00 6D 00
  // L.e. .p.a.r.a.m.
  E8 00 74 00 72 00 65 00 20 00 76 00 61 00 75 00
  // ..t.r.e. .v.a.u.
  74 00 20 00 7B 00 30 00 7D 00 )
```

```
  // t. .{.0.}.
  IL_0005:  ldc.i4.1
  IL_0006:  newarr     [mscorlib]System.Object
  IL_000b:  stloc.0
  IL_000c:  ldloc.0
  IL_000d:  ldc.i4.0
  IL_000e:  ldarg.0
  IL_000f:  box        [mscorlib]System.Int32
  IL_0014:  stelem.ref
  IL_0015:  ldloc.0
  IL_0016:  call       void [mscorlib]System.Console::WriteLine(
                       string, object[])
  IL_001b:  ret
} // end of method Delegates::FonctionDeDelegation
```

On peut voir que ce code est finalement très proche du code source Delphi même s'il est découpé en étapes élémentaires plus nombreuses. Il est en tout cas très lisible et facilement compréhensible, bien plus que le désassemblage d'un exécutable binaire Win32.

Le code Delphi correspond au désassemblage ci-dessus est :

```
procedure FonctionDeDelegation(i : integer);
begin
 Console.WriteLine('Le paramètre vaut {0}',[i]);
end;
```

L'ensemble des métadonnées et du code IL permet l'inspection d'un exécutable .NET depuis l'extérieur, c'est-à-dire en inspectant le fichier disque.

Il est possible d'avoir accès aux mêmes informations depuis l'intérieur d'une application en utilisant le système dit de réflexion qui correspond au système des RTTI (Run Time Type Information) de Delphi. Le système de réflexion joue un rôle essentiel dans l'interopérabilité des langages au sein de .NET.

On notera que la présence du code IL et des métadonnées dans les exécutables .NET pose un problème vis-à-vis du *reverse engineering* puisqu'à l'aide d'un simple utilitaire du framework, on peut obtenir une source modifiable et re-compilable à l'aide d'un autre outil lui aussi fourni dans le framework. Il n'y a plus besoin d'être un hacker de haut niveau pour percer les secrets d'une application existante. En réalité, il est possible de protéger le code .NET de plusieurs façons. En effet, le framework met à disposition un système de signature qui interdit la modification de l'exécutable ; par ailleurs, des outils appelés « obfuscateurs » rendent le code CIL illisible. Nous aborderons ces aspects ultérieurement.

Assemblages et manifestes

Un assemblage est l'unité de base d'un code .NET pouvant être déployé. Il est constitué de modules, d'un manifeste et de ressources. Ce terme lève l'ambiguïté qui régnait dans le monde COM lorsque Microsoft parlait de composants aussi bien à propos de classes que de DLL. Sous .NET, l'assemblage est un exécutable qui entre dans une logique *plug'n'play* accompagné de son manifeste, une description de l'assemblage. Le manifeste peut être intégré à l'exécutable ou bien être fourni sous la forme d'un fichier PE externe décrivant alors un ensemble d'exécutables.

Physiquement, un assemblage .NET peut être soit un EXE soit une DLL et son manifeste indique quels autres assemblages il utilise, leur version, leur culture.

À la différence du modèle COM qui faisait une utilisation débridée des GUID auxquels il fallait faire référence sans cesse dans les applications, .NET propose une notation plus humaine en permettant le référencement d'une classe par le biais de son nom et de son espace de nom. Toutefois, ce principe ne garantit plus l'unicité universelle des GUID. Pour combler cette lacune, un assemblage qui sera partagé se doit d'être signé avec un couple de clés publique et privée. Le résultat du *hashage* de l'assemblage avec la clé privée servant à la signature est stocké dans le fichier PE de l'assemblage ; la clé publique permet ensuite de vérifier la signature d'un assemblage. Cette vérification est d'ailleurs effectuée automatiquement par le CLR, ce qui est l'un des points importants de la sécurité sous .NET. Alors que sous Windows, certaines attaques (virus, vers, piratage) pouvaient se faire par la méthode dite « de substitution de code » en modifiant, par exemple, le code d'une DLL ou en fournissant tout simplement une fausse DLL de même nom, la signature cryptographique d'un assemblage et la vérification systématique celle-ci par le CLR interdit toute manipulation de ce type. Si tout assemblage possède un nom, le fait de le signer numériquement lui confère ce qu'on appelle un nom fort (*strong name*).

Le ramasse-miettes

Le ramasse-miettes ou GC (*Garbage collector*, et moins poétiquement « éboueur » en français) est un mécanisme essentiel car il prend en charge la libération automatique de la mémoire. Si certains développeurs sont réticents à cette forme d'automatisme, il s'agit plus du poids des habitudes que de réelles objections techniquement fondées. D'autant plus que la libération de la mémoire et des ressources en général est l'une des causes les plus

répandues de bogue dans les langages qui laissent cette charge au développeur !

D'ailleurs, un tel système de gestion automatique de la mémoire a fait son entrée il y a déjà quelques temps sous Delphi… Il s'agit de la gestion des chaînes longues et des tableaux dynamiques dont l'allocation et la libération sont gérées par le compilateur. Certes, il ne s'agit pas d'un réel ramasse-miettes puisque sa portée n'est pas globale (elle ne concerne que les chaînes, tableaux dynamiques ainsi que les interfaces) mais de la mémoire est malgré tout allouée automatiquement et détruite de la même façon sans que personne n'y voit d'inconvénient, bien au contraire. Combien d'applications en C ont-elles présenté de dysfonctionnements ne serait-ce que sur les allocations et les libérations des `PChar` ? Aucun problème de ce type n'est arrivé à un développeur Delphi utilisant les chaînes longues… Aujourd'hui, le ramasse-miettes généralise la même sérénité de développement à tous les objets. Qui s'en plaindrait !

De fait, il ne faut pas avoir peur du ramasse-miettes car il apporte au contraire un niveau de fiabilité supplémentaire aux applications. Il ne faut donc pas le voir comme une façon de déposséder le développeur d'un contrôle absolu sur ses programmes mais bien comme un élément de sécurité apportant une garantie contre les pertes de mémoire.

Quelques avantages de .NET

Comme nous le disions en début de cet article, .NET a été conçu dans l'optique d'une informatique intégrée et « agile ». Cette agilité n'est pas seulement un concept commercial, Microsoft a réellement tenté de simplifier les choses, même pour le développeur.

Simplicité de développement

.NET intègre de nombreux comportements et encapsule beaucoup de technologies éparses. Il offre ainsi un cadre cohérent à l'ensemble du développement des applications. Les méthodes restent les mêmes qu'il s'agisse de créer un formulaire Web ou une application fenêtrée pour Windows.

De grandes simplifications ont été faites dans l'utilisation et la création de composants COM, dans la création de serveurs ou de clients de services Web…

Le CLI, qui impose certains comportements aux langages .NET, garantit une sécurité accrue et élimine la possibilité de certains bogues classiques. La gestion d'un ramasse-miettes mémoire par exemple supprime la possibilité d'erreurs dans l'allocation de la

mémoire et des ressources, cause fréquente de dysfonctionnement des applications sous Win32.

Simplicité du déploiement

À l'heure actuelle, sous Windows, le déploiement d'un composant COM par exemple impose de nombreuses manipulations parmi lesquelles l'enregistrement dans la base de registres. Tout repose sur cette dernière et le moindre problème rend tout ou partie des logiciels inutilisables. Microsoft a enfin compris la fragilité de ce système et sous .NET, le déploiement d'un assemblage se fait par simple copie dans un répertoire ! Grâce aux informations de sécurité et aux métadonnées, le CLR s'occupe de tout et charge le bon module. Il sait même identifier les assemblages référencés par un exécutable et les télécharger à la demande, ce qui permet de proposer des applications de faible taille dont les modules seront chargés depuis Internet ou d'un intranet, selon les besoins.

L'utilisation du cache par Shadow Copy simplifie aussi la mise à jour des fichiers : n'étant pas utilisés directement, il est possible de mettre à jour un EXE ou une DLL sur disque sans rebooter, même si le fichier est en cours d'exécution.

Simplicité de la gestion des versions

À l'aide des informations de versions stockées dans le manifeste, des définitions des espaces de nommage et des clés publiques de signature des assemblages, le CLR est capable d'isoler un code donné sans ambiguïté.

Il est ainsi tout à fait possible de diffuser plusieurs versions d'une même DLL portant le même nom de fichier qui peuvent pourtant coexister et s'exécuter en même temps sans que cela ne crée de conflit. Ce que certains appellent « l'enfer des DLL » sous Windows n'existe plus sous .NET.

Le moyen de faire cohabiter plusieurs versions du même fichier portant le même nom physique ne relève pas de la magie mais du GAC (Global Assembly Cache – cache global des assemblages). Puisque le CLR n'exécute pas directement un fichier mais sa copie compilée en binaire dans le cache, il suffit de forcer l'installation dans ce cache des différentes versions du fichier en utilisant GACUTIL, un utilitaire du framework. Une fois dans le cache, l'assemblage est reconnu par .NET grâce aux informations du manifeste, le nom physique du fichier n'a plus d'importance. Deux DLL portant le même nom ne pourront pas entrer en conflit et les logiciels les utilisant ne connaîtront aucun problème puisque le CLR appellera la bonne version référencée dans ces logiciels…

Côte à côte…

Un peu de romantisme dans ce monde de brutes avec .NET ? Non, mais cette façon de pouvoir exécuter en même temps plusieurs versions d'une même DLL sans qu'il y n'y ait de conflit s'appelle « side by side execution » ou exécution côte à côte….

Conclusion

La présentation du framework .NET dans sa totalité et par le menu réclame bien plus qu'un article, il faudrait un, voire plusieurs livres. Vous en trouverez d'excellents notamment chez Eyrolles et à commencer par ceux que l'auteur a publiés chez cet éditeur (un peu de pub ne peut pas nuire !). Si vous connaissez déjà le framework, ce court article n'aura été qu'un bref rappel, si vous ne le connaissez pas du tout, l'auteur espère vous avoir permis de survoler ces principales fonctions afin que certains termes ou sigles qui apparaîtront plus loin dans cet ouvrage ne vous soient pas étrangers.