Capitolo introduttivo
Il framework .Net

Il framework .Net è un ambiente per lo sviluppo ed il run time di applicazioni che costituisce la base di tutte le tecnologie recenti e si presume future di casa Microsoft. Nasce in via ufficiale all'inizio degli anni 2000 con l'intento di fare un po' un punto e a capo nel caotico mondo della programmazione. La spinta data da Internet in particolare e dall'emergere prepotente di nuove tecnologie esigeva una risposta chiara forte ma, soprattutto, efficiente. .Net è quindi questa risposta, una piattaforma potente e ad alta produttività orientata certamente allo sviluppo di applicazioni tradizionali ma con una fortissima orientazione al Web, all'integrazione con i database, alla gestione di contenuti XML ecc.. Il tutto fortemente integrato nelle sue componenti. Uno sforzo davvero in grande stile che ha concentrato molte risorse della software house di Redmond. ma i risultati, a parer mio, sono stati eccellenti. Sigle come DNA, MFC ecc... appartengono per lo più al passato. Non che si trattasse di tecnologie di scarso valore, tutt'altro. Ma, soprattutto se si pensa allo sviluppo Web oriented, mancava qualche cosa di unificato, di realmente robusto, che unisse potenza ad una certa semplicità sia in termini di modellazione che di mera stesura del codice. Ora questo qualche cosa c'è. Il miglior modo di pensare .Net è comunque che esso non è valido perchè i suoi predecessori non lo erano ma perchè è stato pensato dall'inizio in modo coerente e ambizioso.

.Net è costituito fondamentalmente, o meglio da un punto di vista pratico, dell'unione di 3 parti:

- una serie di tools per lo sviluppo e la gestione di applicazioni multi purpose
- una potente libreria di classi pronte all'uso
- un ambiente run-time per l'esecuzione dei programmi

Lo scopo che Microsoft perseguiva (insieme evidentemente a quelli commerciali peraltro irraggiungibili senza mettere in pista un prodotto più che valido e questo ultimo è un punto che tanti dimenticano...) è quello invogliare uno sviluppo software razionale che segua linee guida standard e che permetta una forte cooperazione tra le varie entità in gioco. La storia di .Net comincia di fronte al mondo nel 2002 quando fu distribuita la versione 1.0 che ottenne curiosità interesse, consensi e le solite critiche che accompagnano i nuovi prodotti destinati ad avere un certo impatto, specialmente se arrivano da Redmond. L'adozione di questa nuova tecnologia fu nel complesso tiepido soprattutto nei primissimi mesi ma, poco alla volta, essa cominciò a far breccia anche per l'imponente spinta proveniente dalla casa madre. Importante, anche da un punto di vista prettamente mediatico, fu l'introduzione di un linguaggio di programmazione del tutto nuovo, C# e anche le profonde modifiche fatte al caro vecchio Visual Basic, stravolto quasi dalle fondamenta e divenutato un linguaggio a oggetti. Alle legioni di programmatori in VB fu dato ampio supporto affichè non si sentissero in qualche modo abbandonati in un mondo, quello apputo a oggetti, per molti di loro quasi ignoto. Nel 2003 debuttò la versione 1.1 che rimediava a qualche debolezza e qualche bug della precedente mentre pesanti cambiamenti si ebbero nel 2005 con la versione 2.0 che introduceva novità importanti come le funzioni anonime ed i generics. Successivamente, siamo nel 2006, arrivò la versione 3.0 e .Net cominciò a diventare veramente un prodotto non solo maturo ma anche molto sofisticato con una diffusione in rapida espansione.Nel 2007 si passò alla versione 3.5 e il 2010 sarà l'anno di un altro grosso passo in avanti con .Net 4.0.

Ma quali sono i concetti cardine di .Net? Fondamentalmente Microsoft si è prefissata lo scopo di creare un ambiente che favorisse lo sviluppo pienamente object oriented, che semplificasse la vita allo sviluppatore, ad esempio con un miglior controllo delle versioni e della memoria, consentendogli di scrivere codice più sicuro e fornendo strumenti all'avanguardia. Si sono inoltre definite delle linee di sviluppo che hanno fortemente facilitato l'inserimento nel filone .Net di numerosi altri linguaggi. Tra questi segnalo, per interesse personale IronPython e IronRuby, porting nel mondo .Net di Python e Ruby, forse i due linguaggi di scripting più evoluti e conosciuti, ma ce ne sono veramente tanti vale la pena segnalare la "traduzione" nel mondo di .Net del Cobol e del Fortran, pietre miliari nella storia della programmazione, ma anche Ada, Delphi, oppure la nascita di nuovissimi linguaggi decisamente interessanti come Boo, Cobra, Nemerle. Proprio la possibilità che un gran numero di idiomi che potessero collaborare sotto un ombrello comune era una delle finalità principali in fase di progettazione del framework e direi che è stata pienamente raggiunta.

E veniamo ora ad analizzare il framework da un punto di vista per vosì dire anatomico e fisiologico.

C# - Capitolo introduttivo – Il framework .Net

Gli strumenti ed i tools messi a disposizione da MS sono veramente numerosi, anche solo semplicemente considerando ciò che l'SDK base di .Net ci mette gratuitamente nelle mani. Di essi tuttavia parleremo quando sarà utile farlo, per ora è importante capire quale è il ruolo di .Net.

Lo sviluppo di un software avviene normalmente secondo questo schema almeno per i linguaggi compilati:

La prima fase, la scrittura del codice è normale per tutti i linguaggi anche nell'ambito del nostro framework. Va sottolineato come per quanto riguarda i linguaggi da essa supportati ovvero C#, VBNet e da ultimo anche F#, il linguaggio funzionale ispirato a OCaml, Microsoft mette a disposizione l'ottimo ambiente di sviluppo Visual Studio, disponibile anche gratuitamente nella comunque valida versione Express; tuttavia potete usare l'editor che più vi aggrada, ovviamente i programmi sono pur sempre solo dei files testuali pur se è evidente che strumenti evoluti usati correttamente garantiscono una produttività di altissimo livello. Arriviamo ora alla fase di verifica del codice, che viene svolta dal compilatore del linguaggio prescelto. A meno di errori l'output di questo strumento è un file che per semplicità esplicativa supponiamo sia normalmente quella di un normale eseguibile, ovvero .exe. Si possono anche creare delle DLL ma, insomma il discorso non cambierebbe e comunque per ora ci concentriamo sugli eseguibili. L'impressione quindi è che ci troviamo di fronte a qualche cosa di conosciuto ma in realtà il processo di compilazione genera qualche cosa che non è esattamente un eseguibile come lo intendiamo noi. Ha una estensione .exe ma se provate a farlo girare su di un computer sul quale non vi sia il framework .Net installato non ne otterrete nulla. Quindi c'è qualche cosa di diverso dalla semplice generazione di un exe così come lo pensiamo comunemente. In effetti ogni compilatore .Net non genera un file direttamente eseguibile dal sistema operativo ed è questo il motivo per il quale non è possibile la sua esecuzione senza l'intermediazione del framework. Quello che otteniamo dal processo di compilazione è un file in formato **PE** (**Portable Executable**) che contiene una gran quantità di informazioni che saranno usate per il runtime. In particolare esso è costituito da una parte di codice in Intermediate Language (IL che altrove è indicato come CIL o MSIL) e da un **manifesto** del cui scopo riparleremo. In fase di creazione viene importata dal framework una funzione che si chiama *_CorExeMain* che fa da tramite tra il sistema operativo e il runtime di .Net. Quando l'utente lancia l'esecuzione del programma il loader del Sistema Operativo carica il .exe e cerca l'entry point. Di qui viene rimandato alla funzione _CorExeMain il quale manda in esecuzione il codice IL nel quale è stato tradotto il nostro programma.

Come si vede, quando scriviamo codice siamo ad alto livello, con una astrazione piuttosto marcata rispetto alla macchina. Il codice IL non è eseguibile nativamente ma.Net come detto mette a disposizione tutto ciò che serve per la nostra applicazione. In particolare in questo stadio entra in azione un JIT (= Just In Time Compiler) che si preoccupa di prendere il nostro codice IL e trasformarlo in eseguibile per l'architettura sottostante. Per ottimizzare il lavoro il JIT compila al volo solo le parti che sono effettivamente chiamate in causa nel momento dell'esecuzione, ecco in questo modo chiarito il concetto di just-in-time o meglio, di compilazione al volo. Ovviamente il codice compilato dal JIT viene posto in cache in memoria così che non sia necessario ricorrere a più compilazioni della stessa porzione di codice. In realtà nell'ambito del framework esistono almeno 3 Jitters, uno "normale" appena descritto ed un altro detto "**EconoJit**" più veloce ma che genera codice meno ottimizzato e quindi di norma più lento in esecuzione ed **il PreJit** che compila tutto il codice in un passaggio solo. In breve se il primo passaggio nella figura precedente tutto sommato resta immutato il secondo, tra compilazione ed esecuzione, andrebbe spezzettato in più fasi ad esempio come nell'immagine seguente tenendo tuttavia presente, vale la pena ribadirlo, che il programmatore e soprattutto l'utente non avranno alcun coinvolgimento diretto in quello che combina il compilatore e su quanto accade in fase di esecuzione:

Compilazione
Creazione del codice IL
Richiesta di esecuzione
Compilazione al volo (JIT)
Esecuzione da parte del S.O

A questo punto risulta fondamentale il ruolo dell'ambiente run time provvisto da .Net che prende il nome di Common Language Runtime (di seguito CLR). Questo componente è il vero fulcro del framework pensato da Microsoft ed una utile piattaforma comune a qualunque linguaggio. Esso è in grado di gestire l'Intermediate Language che sia stato generato secondo lo schema proposto qui sopra pertanto e questa è la chiave di tutto il discorso, qualunque compilatore produca IL siffatto vedrà il codice da esso creato funzionare tranquillamente con l'aiuto del run-time. Ecco spiegato perchè, o meglio come .Net è aperto verso qualuque linguaggio, come in fondo ci suggerisce quel "Common" messo come prima parola per il CLR; perchè esso fa girare IL senza riguardo al linguaggio di provenienza. Il contributo di questo strato per il runtime si estrinseca in tutta serie di servizi che vengono messi a disposizione per il codice IL. In particolare esso si preoccupa di astrarre il programmatore da alcune importanti questioni tra le quali segnalo le più significative:

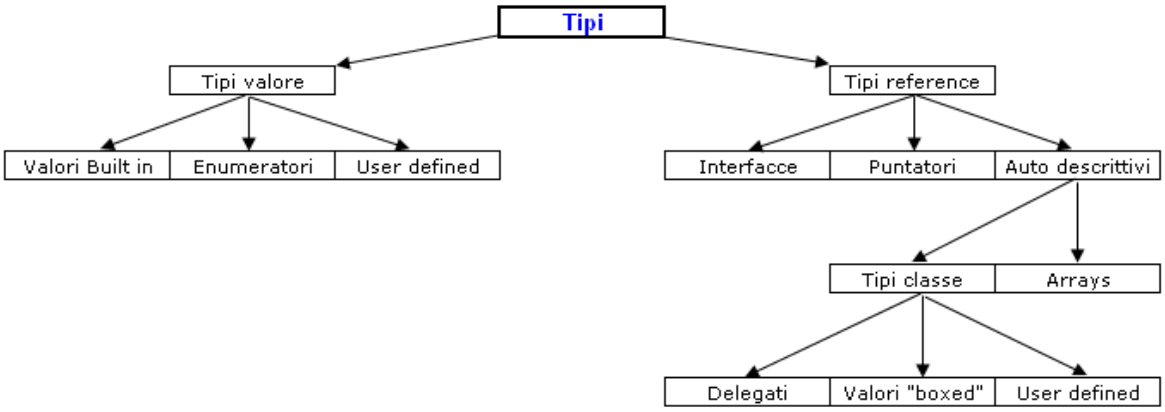
- gestione della sicurezza (molto importante, ne parleremo a parte)
- gestione dei thread
- gestione della memoria
- gestione delle eccezioni
- gestione delle operazioni di pulizia dello heap (garbage collection)

Tutto questo ed anche altro è controllato direttamente dal CLR. Quindi non siamo di fronte solo ad un run-time inteso come esecuzione, sarebbe stato forse un po' poco, ma ad un insieme di servizi forniti al software. Il codice che gira sotto il diretto controllo del CLR viene definito "managed" (appunto "gestito" ma preferisco usare il più comune termine inglese). Per esempio usando codice IL managed non sarà più necessario liberare manualmente la memoria occupata da oggetti non più utilizzati perchè se ne occuperà il CLR; oppure le eccezioni saranno intercettate attraverso meccanismi del runtime stesso con un sistema comune a tutti i linguaggi che seguono le specifiche di .Net. E sappiamo quanto sia importante. Con l'avvento delle moderne CPU multi core, una gestione avanzata e coerente dei thread. In sezioni appositamente definite è tuttavia possibile far girare codice unmanaged, a rischio e pericolo del programmatore.

La generazione di IL è condizione necessaria per ricevere le attenzioni del CLR. Ma non basta. Bisogna ora presentare due sigle per così dire magiche CTS e CLS. La prima sta per Common Type System. Evidentemente si tratta di una serie di regole che definiscono i tipi ai quali devono appartenere le variabili usate nei programmi. La parola Common ci fa per prima cosa capire che si tratta di una struttura che faciliterà la cooperazione tra quanti (linguaggi) ne seguiranno le linee guida. Proprio la cooperazione tra linguaggi è obiettivo primario del CTS il quale mantiene l'impostazione object oriented tipica del mondo .Net. Il CTS prevede due gandi famiglie di tipi: i reference types (tipi per riferimento o tipi di riferimento) e value ypes (tipi valore). Di essi parleremo in dettaglio nel capitolo 3 della sezione dedicata a C# dove verranno evidenziate le dinamiche peculiari nell'utilizzo delle due tipologie. In questa sede ci basta sapere che sono profondamente diversi e che è possibile passare da un tipo all'altro attraverso il meccanismo di boxing e unboxing di cui parleremo nello stesso capitolo.

L'immagine che segue fornisce una visione di massima della suddivsione dei tipi a livello così come intesa da .Net. Premesso che i tipi in se stessi saranno più chiari più avanti è evidente la netta suddivisione tra le due categorie, che pure hanno una origine comune. Il concetto che qui è fondamentale comprendere è che questa figura illustra la base comune a tutti i linguaggi che ambiscono ad essere.Net compliant. Lo so, il concetto della interoperabilità è ripetuto alla nausea ma credo sia utile far vedere quali sono i punti in cui questa convergenza, o meglio convivenza è stata realizzata. Se avete intenzione di scrivere il prossimo grande linguaggio che si basa su .Net questi concetti vi serviranno, che diamine! Lo schema che segue è simile ad altri che potete trovare sulla rete.

.



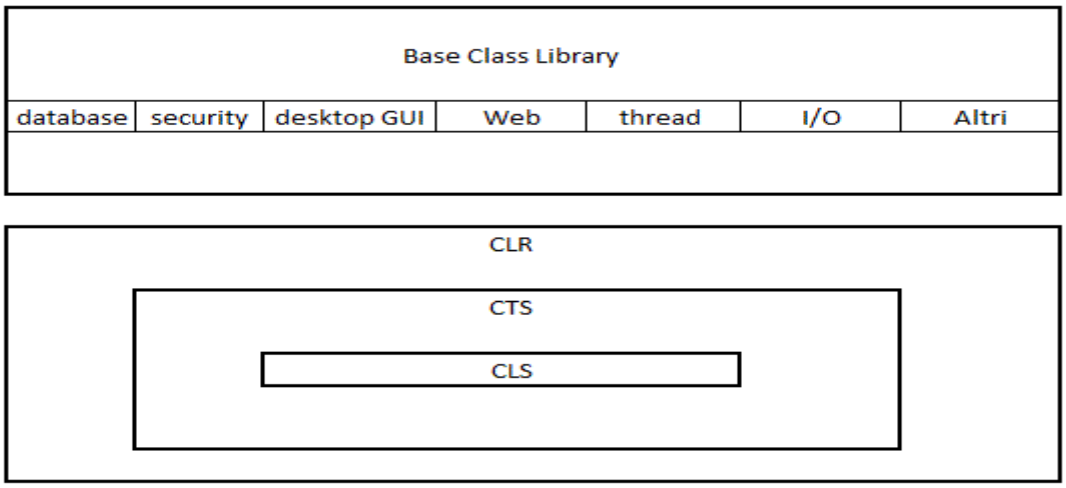
L'immagine qui sopra identifica le due grandi famiglie di tipi. Vale la pena sottolineare come i puntatori in stile C++ non fanno parte del CTS (le loro veci vengono svolte dai delegati) mentre esiste un tipo di managed pointers che è possibile usare nell'ambito CTS. Ne riparleremo. Essere conforme alle specifiche è necessario anche in questo caso per essere compatibili con le richieste del CLR.

Più complessa è la comprensione del secondo acronimo che abbiamo dianzi citato ovvero CLS che sta per Common Language Specification. Eppure si tratta di qualche cosa di concettualmente molto semplice; il CLR è un terreno comune di conversazione tra linguaggi, nonchè uno strato che si pone come intermediario tra il programma e la macchina. Ebbene è necessario definire le regole affichè questo colloquio possa funzionare, quindi stabilire cosa ciascun elemento può esporre e come deve esporlo affiche sia comprensibile. Anche in questo caso siamo di fronte ad una serie di regole che garantiscono l'interoperabilità tra linguaggi. Ovvero si tratta di un insieme minimale (e sottolineo quel "minimale"), di regole alle quali devono rispondere i linguaggi .Net compliant. Questo non è limitativo dei linguaggi stessi: essi possono supportare tutti i tipi di dati e le istruzioni che preferiscono l'importante è che supportino anche quelle che fanno parte del CLS e, qualora sia necessario interagire con altri linguaggi, espongano, almeno nelle parti che dovranno essere comuni, solo le funzionalità riconosciute dal CLS. Questo non si riferisce ovviamente alla grammatica interna dei singoli linguaggi. Un esempio tipico che troverete un po' ovunque su questo argomento è la concatenazione di stringhe che viene ottenuta con "+" in C# e con l'operatore "&" in VBNet. Il codice IL generato è praticamente identico perchè l'operazione di concatenazione è comunque accettabile. Il CLS con le sue regole impatta anche sul CTS il quale accetta una tipologia di dati più ampia rispetto allo stretto sottoinsieme che il CLS accetta per tutti i linguaggi. Ad esempio il tipo enum fa parte del CTS ma non è riconosciuto come universale dal CLS. Oppure Uint32, l'intero senza segno 32 bit non è CLS compliant in quanto non tutti i linguaggi hanno un tipo equivalente. Esiste una complessa tabella, disponibile su MSDN, che spiega, una per una le regole che compongono il CLS. Queste sono molto ampie e coinvolgono sia i dati in se stessi sia la loro nomenclatura, ad esempio non è ammissibile che due variabili differiscano tra di loro solo per il "case" (cioè maiuscolo / minuscolo) dei loro identificatori quindi "gatto" e "Gatto" sono due variabili diverse, ma non CLS ammissibili. Altra regola è l'esclusione dei puntatori dal mondo Common Language Specification. Insomma siamo di fronte ad un tentativo di mettere un po' di ordine tra le features dei linguaggi senza peraltro azzopparne le potenzialità.

Giusto per completezza di informazione segnalo che CLS e CTS sono parte del CLI, Common Language Infrastructure, ovvero l'insieme globale delle caratteristiche definite per l'interoperabilità dei linguaggi in ambito .Net. Si tratta di un insieme di specifiche standardizzate ISO ed ECMA. Ancora una volta il concetto cardine che giustifica l'esistenza del CLI è l'interoperabilità. Il CLR ne è la rappresentazione pratica.

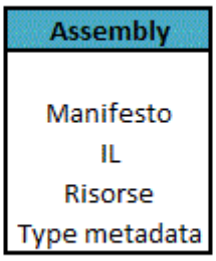
Abbiamo compreso perchè il CLR, ovvero la libreria di runtime è importante e quali sono le regole per goderne i vantaggi. Tuttavia .Net non è solo questo e un aspetto che abbiamo soltanto sfiorato finora è la Base Class Library (di seguito BCL) ovvero l'insieme di classi predefinite che accompagnano .Net e di cui possiamo ampiamente usufruire lavorando in modalità managed. Lo scopo di simili librerie, come le vecchie MFC o le OWL di casa Borland, è sempre quello di sollevare il programmatore dalla necessità di creare o ricreare strumenti per i vari problemi che si incontrano nella programmazione reale fornendo una base comune che viene incontro a varie necessità che si vada dalla gestione di contenuti XML, all'accesso ad un database, all 'uso spinto della grafica. La libreria fornita da Microsoft è amplissima ed offre utili servizi e features praticamente per ogni possibile problematica dello sviluppo. Anche in questo caso la

BCL può, volendo, essere vista come uno strato che viene interposto tra il programmatore e il sistema. Nel complesso una immagine che rappresenta tutte le entità in gioco di cui abbiamo finora parlato è la seguente:



Tale immagine (un po' rozza, in giro se ne trovano di simili un po' migliori) evidenzia come la BCL con i suoi vari servizi offerti si proponga come interfaccia tra il programmatore e il resto dell'ambiente. Net. Non è obbligatorio servirsi della BCL, ognuno si può scrivere le classi che vuole ma è chiaro che essa è stata scritta in maniera da lavorare armoniosamente con le altre componenti ed usarla fa risparmiare evidentemente parecchio tempo trattandosi di codice, in soldoni, ben scritto ed ottimizzato per il framework. La BCL è organizzata in namespaces che, gerarchicamente, possono contenere altri namespace, che a loro volta contengono classi che a loro contengono metodi e proprietà, cose che vedremo meglio parlando di C#. Va rimarcato come i servizi offerti dalla BCL siano veramente tanti e quelli illustrati sono solo una minima parte.

Ora che abbiamo visto le componenti essenziali del framework, restano fuori solo i tools di utilizzo ma quelli li vedremo man mano che serviranno, torniamo indietro ai primi concetti espressi ed analizziamo il processo di creazione di un file eseguibile. Già sappiamo che si tratta di un file in un formato un po' particolare che contiene del codice IL. A fianco a questo abbiamo citato un "manifesto". Questo perchè in realtà quando il compilatore termina la sua opera viene creata una entità che si chiama assembly e che, se generalmente è costituita dal solo nostro eseguibile, almeno per le piccole applicazioni di prova che useremo (come vedremo non è poi così corretto ma in prima istanza questo "quadro" può andare bene), può in realtà essere costituito da più files e più risorse. Un assembly mono-file, permettetemi il neologismo, può essere visto come siffatto:



Mentre un assembly composto da più entità potrebbe avere la parte relativa alle risorse racchiuse in un file esterno, puntare ad una DLL, o ad un un file grafico ecc... e anche comprendere più binari .Net.; in questo caso i vari files binari prendono il nome di moduli e tra di essi ve ne deve essere uno primario che contiene il manifesto nel quale sono elencati tutti i necessari riferimenti. Il manifesto contiene infatti i metadata dell'assembly stesso quindi informazioni ad esempio sui files che ne fanno parte, o relative alle risorse ed ai loro riferimenti, stabilisce le versioni, ad esempio dello stesso framework, necessarie al funzionamento dell'applicazione, informazioni sulla sicurezza ecc... Si tratta di un componente molto importante quindi che può essere abbastanza complesso. Va dato risalto al fatto che il

manifesto contiene informazioni sulle versioni di software (anche DLL) che devono essere usate per l'applicazione. Questa gestione avanzata delle versioni serve a risolvere una annosa questione relativa proprio al fatto che spesso l'installazione di un nuovo programma finiva con lo scontrarsi con problemi per così dire locali a livello di singolo computer e di corredo software dello stesso. Vi ricordate il famoso DLL-Hell (di cui Microsoft ha una certa responsabilità, peraltro)? Beh, ora non vi tormenterà più e più versioni della stessa DLL possono convivere senza pestarsi i piedi. A titolo informativo sottolineo che Net mette a disposizione degli strumenti per indagare sui manifesti di una applicazione. I type metadata invece riportano informazioni accurate sui tipi usati nell'assembly, quindi la loro natura, gli eventuali membri che ne fanno parte se si tratta di classi o strutture e così via. Possiamo assimilare nel complesso i metadati a delle tabelle descrittive. Si tratta di una descrizione precisa e meticolosa. In generale i metadata sono un collante fondamentale per le applicazioni .Net. Infine le risorse indicano le solite informazioni locali relative al file, un esempio classico è l'icona specificata per rappresentare l'eseguibile. Un assembly ha in buona sostanza molti vantaggi in quanto è autodescritto, facilita l'installazione e realmente vale la pena di sottolineare la sua importanza nel dirimere i conflitti tra versioni.

Il concetto di assembly a volte confonde quando ci si confronta con la visuale consueta e rassicurante dell'eseguibile monolitico. In realtà si tratta di un concetto logico abbastanza semplice che è del tutto trasparente all'utente e spesso anche al programmatore, quasi banale quando si parla di assembly costituito da un singolo files ma altrettanto semplice quando le risorse sono molteplici in questo caso l'assembly è l'insieme di queste risorse riunite dal collante del manifesto che si trova nel file primario. E' il passo successivo nella comprensione del concetto di compilazione e può anche essere ignorato perchè non impatta sull'interazione tra utente e macchina almeno in maniera visibile. Però è un concetto importantissimo che lavora pesantemente e silenziosamente dietro le quinte. E costituisce una chiave primaria nella teoria di .Net.

Va comunque detto che lavorando in C# o in altri linguaggi .Net compliant tutti questi discorsi sono a carico del compilatore (altrimenti per scrivere un semplice "Hello World" come quello che trovate nel capitolo 2 ci vorrebbe una giornata di lavoro). Però può essere utile conoscere queste nozioni e sapersi destreggiare tra di esse, in seguito vedremo l'uso di risorse atte ad indagare attentamente su assembly e compagnia.

A proposito di applicazioni costruite nell'ambito .Net non può essere taciuto il concetto di Application Domain. Alla base della loro definizione sta l'idea di **isolamento**, ovvero il fatto che ogni applicazione deve essere in grado di girare senza influenzare involontariamente le altre. Questo era già possibile in precedenza attraverso l'indipendenza dei processi ma a prezzo di un decadimento delle prestazioni in particolare in caso di necessità di comunicazione tra un processo e l'altro quando la totale impossibilità di condividere memoria (il sistema operativo divide rigorosamente i singoli processi) rendeva necessari complessi meccanismi di marshaling con tutte le conseguenze del caso. Usando il modello basato sugli application domain ogni processo può essere contenere (un po' improprio ma rende l'idea) più applicazioni che quindi essendo nella stessa area di memoria possino più facilmente parlare tra di loro per quanto il CLR si occupi comunque di nascondere i dati di una applicazione rispetto alle altre. In questo senso il CLR sta alle applicazioni come il sistema operativo ai processi. Attaverso meccanismi specifici sarà ogni singola applicazione che metterà eventualmente a disposizione i propri dati. Quanto detto è solo un accenno ma questa modo di funzionamento reso possibile dal framework è assai importante e tecnologicamente molto comodo.

Un'altra feature disponibile per tutti i linguaggi che usano cone runtime il CLR di .Net è il cosiddetto Garbage Collector (GC) ovvero lo strumento dedicato a ripulire la memoria (in particolare quella zona identificata come heap, come vedremo) da quegli oggetti non più in uso. La presenza di questo strumento viene incontro alla necessità di rilasciare la memoria non più utilizzata in modo da prevenire i memory leaks e aiutare a prevenire situazioni di errore. Molti programmatori "giovani" di oggi probabilmente non avranno il polso del problema in quanto una gestione automatizzata della memoria è comune a quasi tutti i linguaggi che usiamo. In passato la questione era però spinosa e veniva affrontata principalmente con due tecniche. In C++, per esempio, questo lavoro era svolto manualmente dal programmatore il che è certamente fattibile è può essere reso molto efficiente ma è un approccio estremamente complesso ed esposto a bug insidiosi, nonostante le migliorie nel supporto ai programmatori dei vari strumenti di sviluppo. L'approccio COM invece prevedeva la presenza di una sorta di "contatore di riferimenti" ovvero, grossolanamente, ogni oggetto manteneva una lista di contatti da parte dei vari client autodistruggendosi quando questo valore era pari a zero. Questo è senza dubbio interessante ma prevede una precisa cooperazione tra le varie entità che non è così scontata e banale.Ecco quindi che la risposta in .Net come in Java ecc...è l'introduzione di una modalità gestita dal runtime. Il garbage collector di .Net funziona in modo eccellente nel CLR anche perchè l'Intermediate Language ne favorisce l'utilizzo. Non si tratta di un pensata esclusiva di casa Microsoft come detto, l'idea è anzi piuttosto datata, ma sicuramente l'interpretazione che ne è offerta è di qualità. Ovviamente questo si

realizza utilizzando codice managed, se si comincia ad infarcire il codice di puntatori “vecchio stile” questo ed altri vantaggi vengono persi.

Spendiamo solo un attimo relativamente al discorso delle prestazioni della macchina virtuale che sono di ottimo livello e che possono essere ulteriormente perfezionate attraverso i numerosi strumenti di profiling che aiutano ad individuare le inefficienze del proprio codice. Sono stati fatti vari raffronti con la JVM, Java Virtual Machine che è considerata la piattaforma più direttamente rivale di .Net. Sinceramente non mi sento di esprimere sentenze, anche considerando che le mie esperienze con Java sono troppo datate per essere significative in questo senso, lasciando l’approfondimento alla consultazione dell'ampio materiale disponibile in rete a cui rimando anche per analisi prestazionali relative a .Net aggiornate ed approfondite, suggerendo di allargare quanto più possibile la ricerca senza fermarsi ai primi esiti, non tutti i siti sono “sinceri”.

Come abbiamo accennato in precedenza alcuni linguaggi come Ruby e Python sono entrati a far parte del mondo .Net tramite le loro incarnazioni IronRuby e IronPython. Si tratta di linguaggi dinamici, profondamente diversi in questo senso da C# che, come vedremo ha una tipizzazione statica. A supporto dei linguaggi dinamici il nostro framework è stato dotato del DLR, ovvero Dynamic Language Runtime che assolve in realtà un duplice compito: da un lato consente l’interfacciamento del CLR con i linguaggi citati e altri (siano noti come LISP e SmallTalk o meno come Groovy o l’ottimo Cobra), dall’altro, come vedremo, permette di inserire caratteristiche dinamiche in linguaggi per natura statici, proprio come C#. La definizione testuale di Microsoft per questo progetto (che è Open Source) è la seguente: “The Dynamic Language Runtime (DLR) is a set of libraries built on the Common Language Runtime to support dynamic language implementations on .NET.” Il DLR si pone come strato intermedio tra i linguaggi dinamici ed il CLR; per questo strato **devono passare** i linguaggi dinamici mentre ad esso **possono accedere** quelli statici. Questa è una novità che troveremo ed approfondiremo parlando delle nuove caratteristiche di C# 4.0. Ulteriore conseguenza è un elevato grado di interoperabilità tra linguaggi aventi natura diversa il che è positivo anche in considerazione del buon successo che stanno riscuotendo alcuni dei citati linguaggi di scripting. Alcuni puristi non vedono di buon occhio le nuove caratteristiche dinamiche di un linguaggio originariamente statico come C#. Sinceramente li lascio volentieri alle loro elucubrazioni e abbraccio ancor più volentieri le nuove interessanti caratteristiche che questo nuovo approccio consente.

Terminato il discorso per così dire funzionale intorno a .Net parliamo ora, almeno in via introduttiva, degli aspetti applicativi. Certamente l’idea orginale del progetto guardava al mondo desktop su piattaforma Windows (naturalmente!) e al mondo Web come target principali. Tuttavia questa piattaforma è stata implementata per non tralasciare nulla che possa aiutare il lavoro dei programmatori. Ecco quindi che se, per esempio, vi chiedete se esistono delle features verso il mondo mobile la risposta non può che essere affermativa. La cosa è ovvia se si pensa che Microsoft rilascia un sistema operativo espressamente dedicato ai cellulare ed agli smartphone, il controverso Windows Mobile (ora però è arrivato Windows 7 anche su questi dispositivi e credo che avrà un ottimo impatto). Ecco allora che possiamo parlare di .Net Mobile (qualche volta chiamato, a mio avviso un po’ impropriamente, Microsoft Mobile Internet Toolkit) un ampio set di strumenti e controlli specificamente pensati per il promettente e dinamico mondo mobile. Si tratta di un aiuto importante per chi sviluppa viste le problematiche peculiari e critiche che si incontrano, le più banali e macroscopiche sono la ridotta area di visualizzazione e le criticità di interfacciamento con l’ambiente operativo. Avremo modo di parlare in via superficiale di questo ramo di .Net che mantiene gran parte delle maggiori caratteristiche citate in precedenza. E non finisce qui. Nell’ambito del framework è disponibile un namespace che contiene una serie di classi atte a favorire lo sviluppo di animazioni e giochi. L’acronimo con cui è noto questo ambito e i tools di sviluppo associati è XNA (strana sigla: 'XNA's Not Acronymed' però è orecchiabile ☺) ed esiste anche un IDE dedicato. Per quanto lo sviluppo di giochi non sia evidentemente una cosa banale siamo di fronte ad un altro ramo molto interessante e che, se non altro, promette di facilitare alcune cose, restando nell’ambito Microsoft e, in particolare, se ci si basa su **DirectX**. E se volete andare oltre, sia pure allargando un po’ l’ambito anche al di fuori di .Net, indagate su cosa sia Microsoft Robotics. Ovviamente non si può sapere e conoscere bene tutto ma occorre concentrarci su ciò che ci serve. Sappiamo però che con .Net saremo più o meno sempre a casa nostra.

Molto importante è ricordare che la versione 4 ci porta direttamente nel mondo della elaborazione parallela per poter sfruttare adeguatamente le nuove CPU multicore. Si tratta di un argomento piuttosto avanzato che affronteremo piuttosto avanti in maniera completa. Esso, si può definire la prossima frontiera nell’ambito della programmazione, apre veramente grandi prospettive che verranno ulteriormente sviluppate nella prossima versione 5.0 attualmente (novembre 2010) solo accennata sui canali dedicati di Microsoft. Il futuro si preannuncia davvero molto interessante e ricco ma l’importante per ora è apprezzare il notevole aiuto che il framework ci può fornire già da adesso per

affrontare questa nuova avvincente sfida (almeno se ci si riferisce al “largo consumo”) con la quale ci confronteremo presto nell’immediato futuro.

Infine giova ricordare, per chi non ama Windows o ha comunque orizzonti più ampi, che esiste il progetto **Mono**, porting nel mondo Linux e Mac del framework. Tramite questo è possibile scrivere programmi ad esempio in C# per il sistema operativo caro a Bill Gates e vederlo comunque girare, teoricamente senza modifiche, anche sulla vostra distribuzione Linux preferita o su Mac. La cosa non è così semplice in realtà e va inoltre considerato che, per forza di cose, Mono è sempre un po' indietro rispetto al progetto ufficiale. Tuttavia è pur sempre un interessante e ben avviato tentativo di allargare il bacino di utenza di .Net e va seguito con curiosità e attenzione specialmente se volete che le vostre applicazioni abbiano come target più di un sistema operativo. Vedremo, molto più avanti in apposita sezione, qualche suggerimento che dovrebbe facilitare la scrittura di applicazioni cross-OS.

In conclusione quali sono schematicamente i punti di forza di .Net? In giro sulla rete e nei vari testi ne troverete parecchie di tabelle e pareri personali che chiariscono questo aspetto; da parte mia sono concorde su questi punti:

- facile (relativamente, si capisce) ed uniforme sviluppo verso l'ambito desktop e quello Web con possibili espansioni anche in altri ambiti (quello ludico ad esempio e ancora di più quello mobile).
- ottima integrazione con i database e in generale con le tecnologie Microsoft ma anche un buon supporto verso XML.
- Ambiente omogeneo e architetturalmente coerente, fortemente inclinato verso lo sviluppo a oggetti che viene largamente favorito dall'impostazione del framework
- Migliorato supporto alla sicurezza in tutti gli ambiti (anche se, a parer mio, c'è ancora un po' di lavoro da fare)
- eccellente integrazione con più linguaggi
- facilità di installazione delle applicazioni
- grande quantità di strumenti e di supporto per lo sviluppo di applicazioni di qualsiasi livello. Molte communities on line e molto materiale (sia pure di qualità non omogenea) facilmente reperibile sul web e in varie pubblicazioni (riviste, libri). Microsoft sul suo sito nel ramo MSDN mantiene disponbile on-line una mole credo ineguagliabile di materiale.
- Sviluppo costante e migliorie continue. Questo è un aspetto da non sottovalutare. Spesso alcuni strumenti, anche di ottima qualità, vengono lasciati molto tempo senza aggiornamenti, magari anche solo per carenza di “mani”, il che se da un lato ne consolida l'uso dall'altro può essere causa di una qualche obsolescenza e rende, paradossalmente, più difficile l'introduzione e l'accettazione di importanti novità. Con .Net questo non succede. La scalabilità insomma è un aspetto importante che viene sfruttato in pieno certamente grazie alle risorse che a Redmond non mancano. Il prezzo da pagare è uno studio continuo al fine di rimanere costantemente aggiornati. Forse pesante ma sicuramente stimolante.