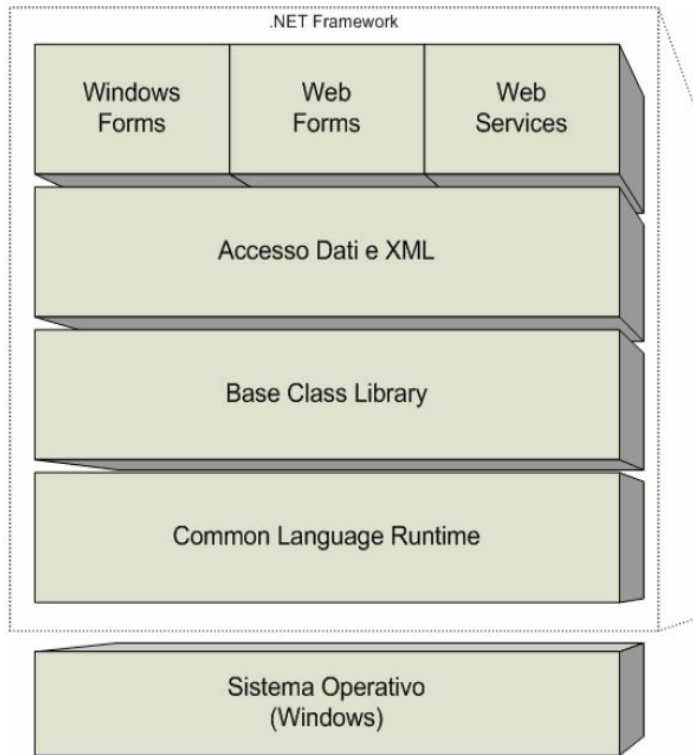


Il .NET Framework

By Dario Maggiari

L'architettura del .NET Framework è riassunta, nel complesso, nella figura seguente:



Il cuore del .NET Framework è costituito dal **CLR** (*Common Language Runtime*) che, secondo lo standard ECMA, prende il nome di **CLI** (*Common Language Infrastructure*). Il termine CLI è senza dubbio semanticamente più adatto a descrivere questo strato del .NET Framework poiché non si tratta di un semplice supporto a tempo di esecuzione ma di un livello che fornisce alle applicazioni managed, cioè le applicazioni che sono direttamente gestite dal CLI, una serie di servizi aggiuntivi.

Il CLR si occupa di gestire l'esecuzione dei programmi scritti per la piattaforma .NET e, nello specifico, si preoccupa di:

- Interfacciarsi, a basso livello, con il sistema operativo evitando di far intervenire il programmatore direttamente sul registry e sulle configurazioni di sistema rendendo più veloce e meno difficoltoso il deployment di un prodotto.
- Gestire la memoria tramite un sistema di garbage collection e introdurre funzionalità di sicurezza gestendo il codice managed cercando di evitare, ad esempio, situazioni di reference dangling e memory leak.
- Gestire le dipendenze.
- Permettere l'interoperability fra i linguaggi managed dal .NET framework.

Il programmatore, in realtà, non interagisce direttamente con il CLR/CLI ma tramite un sistema di librerie detto **BCL** (*Base Class Library*) che mette a disposizione operazioni fondamentali per l'I/O, il trattamento di file raw e strutturati, il trattamento delle stringhe, della connettività, etc...Al di sopra della BCL si trovano le classi per l'accesso ai database e per la manipolazione dei file XML. Infine, a livello di interfaccia, il .NET Framework mette a disposizione una serie di strumenti per la

realizzazione di componenti grafici sia a livello desktop application che a livello web, sia per la definizione di servizi da esportare via web.

CLR/CLI.

Il CLR è costituito da 5 componenti fondamentali:

1. **CIL/IL** (*Common Intermediate Language/Intermediate Language*)
2. **JIT** (*Just In Time*)
3. **CTS** (*Common Type System*)
4. **CLS** (*Common Language Specification*)
5. **VES** (*Virtual Execution System*)

Il codice managed (C#, C++ Managed, J#, VB) è direttamente gestito dal CLR che si occupa di fornire servizi non solo a runtime ma anche in fase di sviluppo.

✓ *A runtime il CLR interviene sui seguenti aspetti:*

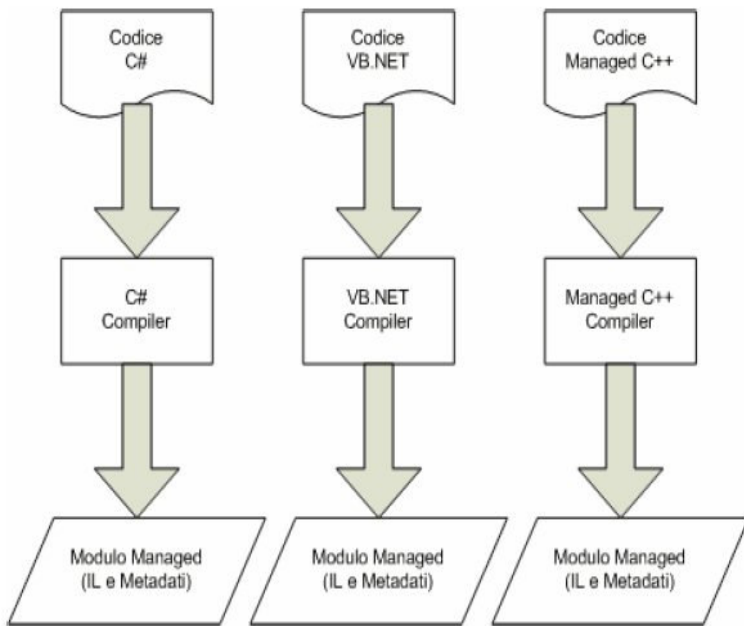
- Gestisce la memoria (garbage collection) istanziando e deallocando oggetti dall'heap.
- Gestisce il ciclo di vita di processi e threads.
- Interviene sulla sicurezza con un sistema dinamico di controllo degli overflow anche grazie al sistema di eccezioni e della tipizzazione forte.
- Interviene dinamicamente sulle dipendenze.

✓ *In fase di sviluppo il CLR interviene sui seguenti aspetti:*

- Gli automatismi introdotti, come ad esempio la gestione della memoria, permettono di scrivere meno codice ma più sicuro e in modo più semplice.
- La tipizzazione forte, la gestione del ciclo di vita, l'introduzione del sistema ad eventi, il binding dinamico e i sistemi di reflection permettono di scrivere meno codice e in modo più semplice.

- CIL/IL

Il vero punto di forza del CLR consiste, però, nella capacità di gestire in maniera unificata diversi linguaggi (C#, C++ Managed, J#, VB) gestendo, a runtime, NON direttamente il codice nativo scritto in C#, C++ Managed, J#, o VB ma codice **CIL** (*Common Intermediate Language*) detto anche **IL** (*Intermediate Language*) che poi viene tradotto ed assemblato in un eseguibile .NET. Quindi, il risultato della compilazione è un *modulo managed* (vedasi figura successiva) cioè aderente alle regole del CLR, anche se è possibile scrivere codice unmanaged che scavalca i



controlli del CLR e per questo è di sua natura “insicuro”. Il risultato della compilazione non consiste esclusivamente in moduli managed ma anche in metadati che descrivono i tipi, i membri e i riferimenti nel codice. Il CLR, però, non lavora direttamente sui singoli moduli ma su **assembly** cioè un raggruppamento logico di moduli managed e un insieme di risorse come file di configurazione, file html, immagini etc... che vanno a costituire un’entità a se stante autodescrittiva che

è “rappresentata” da un file manifesto (*manifest*) che contiene tutti i riferimenti ai file contenuti in esso.

- JIT Compiler

Il codice IL non è eseguibile direttamente dal processore e nemmeno dal CLR poiché esso NON ha funzione di interprete. E’ il compilatore **JIT** (*Just In Time*), detto anche **JITter**, che si preoccupa di tradurre i moduli managed, scritti in IL, in codice nativo eseguibile dalla macchina sulla quale sta girando lo stesso JITter e dunque il programma. Il concetto fondamentale da ribadire è che il CLR non ha funzioni di interprete ma on-demand effettua la traduzione dall’IL al codice nativo. Ad esempio, la prima volta che si invoca un metodo viene compilato il suo codice e viene memorizzato in cache in modo da non dover effettuare una nuova ricompilazione quando esso verrà eventualmente richiamato successivamente. Questo principio si basa sui concetti generalizzati dell’80/20 e del *full-features utilization*: non solo l’80% del tempo di esecuzione è impiegato solo dal 20% delle istruzioni, ma alcuni rami di codice durante l’esecuzione del programma potrebbero essere completamente ignorati. Quindi il CLR risparmia il tempo e l’occupazione di memoria dovute alla compilazione dell’intero CIL dei moduli managed allo startup del programma ed esegue la traduzione on demand conservando il codice compilato per le chiamate successive. Inoltre, il codice può essere compilato indipendentemente dall’architettura sottostante e ottimizzato in base ad essa a patto che esista un CLR (nello specifico, un JITter...) adatto alla piattaforma sottostante. Non solo il codice può essere considerato indipendente dall’architettura ma anche dall’OS poiché è possibile pensare ad un CLR adatto ad interfacciarsi con l’OS utilizzato.

Ad esempio, il progetto *Mono .NET* rappresenta un'ottima implementazione del .NET Framework per il sistema operativo Linux.

- CTS e CLS

Per permettere l'interoperability dei linguaggi gestiti dal runtime è necessario definire un sistema di tipi comune direttamente gestito dal CLR. In effetti, i linguaggi utilizzati nel .NET Framework non hanno ovviamente solo differenze sintattiche, ma hanno peculiarità particolari che differiscono fra loro come, ad esempio, diversi modi di gestire l'ereditarietà, di dichiarare i tipi, di utilizzare eventi ed eccezioni etc...Il CTS (*Common Type System*) definisce un insieme di tipi comune gestito direttamente dal runtime e che permette l'integrazione profonda dei diversi linguaggi di programmazione. In questo modo è possibile scrivere librerie in un solo linguaggio evitando di effettuare il porting o innestare codice C# in porzioni di codice C++ e viceversa.

Il CTS lavora su 2 tipi fondamentali:

- *Value Type:*

Descrivono/mappano tutti quei tipi i cui valori sono rappresentati come una sequenza di bit. Sono solitamente allocati sullo *stack di un thread* e la variabile che rappresenta l'oggetto contiene il valore effettivo dell'oggetto stesso e NON un puntatore ad esso. E' possibile convertire un tipo Value in un tipo Reference e viceversa tramite i meccanismi di Boxing e Unboxing.

I Value Type si suddividono in:

- *Built-in:* float, int, double, etc...
- *User defined:* tipi, anche strutturati, definibili dall'utente.

- *Reference Type:*

Descrivono/mappano tutti quei tipi i cui valori sono rappresentati come la locazione di una sequenza di bit. Questi tipi sono allocati nella memoria dal *managed heap* che si preoccupa di restituire l'indirizzo di accesso alla memoria che fa riferimento ai bit effettivi di un oggetto.

I Reference Type si suddividono in:

- *Built-in.*
- *Self Describing.*
- *Pointer Type:* function pointer, managed type ed unmanaged type.
- *Interface Type.*

Per permettere l'interoperability di codice scritto in linguaggi differenti è necessario che chi fornisce librerie, o progetta un nuovo linguaggio, aderisca allo standard CLS (*Common Language*

Specification) cioè un sottoinsieme del CTS che introduce una serie di regole che definiscono come devono essere generati gli assembly per avere codice “*CLS Compliant*”, cioè interoperabile.

- VES

Il VES (*Virtual Execution System*) ha un ruolo equivalente alla JVM in ambiente SUN poiché carica, linka ed esegue i file eseguibili in formato PE (*Portable Executable*). Il VES è un loader che adempie alle sue funzioni sfruttando le informazioni presenti nei metadati e fornendo esso stesso servizi a tempo di esecuzione come la gestione della memoria, debugging, profiling e sandbox analoghe a quelle di Java per incrementare il livello di sicurezza a seconda della tipologia di codice da eseguire.