ALMA MATER STUDIORUM - UNIVERSITÀ DI BOLOGNA

**FACOLTA’ DI INGEGNERIA**

**CORSO DI LAUREA IN INGEGNERIA INFORMATICA**

**TESI DI LAUREA**

in

Fondamenti di Informatica L-B

PORTING E PROCESSO DI MANTENIMENTO DI APPLICAZIONI JAVA SU PIATTAFORMA .NET

|  |  |
| --- | --- |
| CANDIDATO: | RELATORE: |
| Alessandro Montanari | Chiar.mo Prof. Enrico Denti |
|  |  |

Anno Accademico 2009/2010

Sessione II

# Indice

[Introduzione 4](#_Toc279569530)

[1 IKVM.NET 5](#_Toc279569531)

[1.1 Requisiti 5](#_Toc279569532)

[1.2 Installazione 5](#_Toc279569533)

[1.3 Modalità dinamica 6](#_Toc279569534)

[1.4 Modalità statica 6](#_Toc279569535)

[1.5 Tradurre applicazioni Java in .NET 7](#_Toc279569536)

[1.5.1 Standard Class Library 7](#_Toc279569537)

[1.5.2 Librerie esterne 8](#_Toc279569538)

[1.5.3 Sistema di Class Loading 9](#_Toc279569539)

[1.5.4 Debugging 12](#_Toc279569540)

[1.6 Applicazioni .NET in Java 12](#_Toc279569541)

[1.6.1 Strumento ikvmstub.exe 13](#_Toc279569542)

[1.6.2 Caratteristiche .NET in Java 13](#_Toc279569543)

[1.6.3 Debugging 14](#_Toc279569544)

[2 Ja.NET 15](#_Toc279569545)

[2.1 Requisiti 15](#_Toc279569546)

[2.2 Installazione 15](#_Toc279569547)

[2.3 Modalità statica 16](#_Toc279569548)

[2.4 Compilare applicazioni Java in .NET 16](#_Toc279569549)

[2.4.1 Standard Class Library 16](#_Toc279569550)

[2.4.2 Librerie esterne 17](#_Toc279569551)

[2.4.3 Esecuzione 18](#_Toc279569552)

[2.4.4 Sistema di Class Loading 18](#_Toc279569553)

[2.4.5 Debugging 20](#_Toc279569554)

[2.5 Applicazioni .NET in Java 20](#_Toc279569555)

[2.5.1 Caratteristiche .NET in java 21](#_Toc279569556)

[2.5.2 Debugging 21](#_Toc279569557)

[3 Confronto tra IKVM.NET e Ja.NET 22](#_Toc279569558)

[3.1 Usabilità 22](#_Toc279569559)

[3.1.1 Conversione da Java a .NET 22](#_Toc279569560)

[3.1.2 Applicazioni .NET in Java 22](#_Toc279569561)

[3.2 Completezza 23](#_Toc279569562)

[3.3 Class Loading 23](#_Toc279569563)

[3.4 Deployment 23](#_Toc279569564)

[3.5 Supporto 24](#_Toc279569565)

[3.6 Conclusione 24](#_Toc279569566)

[4 Caso di studio: tuProlog 25](#_Toc279569567)

[4.1 Introduzione 25](#_Toc279569568)

[4.2 Librerie 25](#_Toc279569569)

[4.3 Interoperabilità con Java 26](#_Toc279569570)

[4.4 tuProlog su piattaforma .NET 26](#_Toc279569571)

[4.5 Obbiettivo 27](#_Toc279569572)

[4.6 Piano di verifica 27](#_Toc279569573)

[4.6.1 Test di unità 27](#_Toc279569574)

[4.6.2 Test di accettazione 28](#_Toc279569575)

[4.6.3 Test delle prestazioni 30](#_Toc279569576)

[4.7 Approccio basato su IKVM.NET 31](#_Toc279569577)

[4.7.1 Risultati test di unità 31](#_Toc279569578)

[4.7.2 Risultati test di accettazione 32](#_Toc279569579)

[4.7.3 Risultati test delle prestazioni 33](#_Toc279569580)

[4.8 Approccio basato su Ja.NET 34](#_Toc279569581)

[4.8.1 Risultati test di unità 34](#_Toc279569582)

[4.8.2 Risultati test di accettazione 34](#_Toc279569583)

[4.8.3 Risultati test delle prestazioni 35](#_Toc279569584)

[4.9 Conclusione 35](#_Toc279569585)

[5 Supporto all’interoperabilità per linguaggi .NET 36](#_Toc279569586)

[5.1 Analisi del problema 36](#_Toc279569587)

[5.2 OOLibrary 38](#_Toc279569588)

[5.2.1 Convenzioni 39](#_Toc279569589)

[5.2.2 Interfaccia 39](#_Toc279569590)

[5.2.3 Possibili soluzioni in Java 42](#_Toc279569591)

[5.2.4 Possibili soluzioni in .NET 43](#_Toc279569592)

[5.2.5 Confronto 43](#_Toc279569593)

[5.2.6 Implementazione delle convenzioni 44](#_Toc279569594)

[5.2.7 Implementazione della libreria 46](#_Toc279569595)

[5.2.8 Collaudo 48](#_Toc279569596)

[Appendice 52](#_Toc279569597)

[Appendice A – Esempi di utilizzo di IKVM.NET 52](#_Toc279569598)

[Appendice B – Esempio di applicazione .NET scritta in Java con IKVM.NET 53](#_Toc279569599)

[Appendice C – Esempi di utilizzo di Ja.NET 54](#_Toc279569600)

[Appendice D – Esempio di applicazione .NET scritta in Java con Ja.NET 55](#_Toc279569601)

[Appendice E – Approfondimento *NETFileRunner.exe* 56](#_Toc279569602)

[Appendice F – Esempio di utilizzo di tuProlog su piattaforma .NET 57](#_Toc279569603)

[Bibliografia 61](#_Toc279569604)

# Introduzione

L’obbiettivo principale di questa tesi è analizzare i progetti IKVM.NET e Ja.NET per individuare quale dei due si presti meglio a sostenere un processo di porting di applicazioni Java sulla piattaforma .NET. L’intento è quello di automatizzare il più possibile tale processo per consentire di ottenere la versione .NET di un’applicazione senza, idealmente, apportare alcuna modifica e senza cambiamenti visibili nel funzionamento. La prima parte quindi si concentra sullo studio di questi strumenti analizzandoli nel dettaglio.

Nella seconda parte sono utilizzati gli stessi strumenti per cercare di realizzare una versione per la piattaforma .NET dell’interprete tuProlog, sviluppato presso il Dipartimento di Elettronica Informatica e Sistemistica della Facoltà di Ingegneria dell’Università degli Studi di Bologna. Il lavoro parte da due tesi precedenti in cui è stata svolta una conversione manuale dell’interprete utilizzando il linguaggio C#. Tuttavia in seguito all’impossibilità di sostenere lo sviluppo dell’interprete su due piattaforme distinte è nata l’esigenza di trovare uno strumento che permetta di ottenere la versione .NET con un lavoro minimo potendo anche supportare la conversione di versioni future. Durante questa fase del lavoro l’attenzione si è focalizzata sul funzionamento dell’interprete, cioè la versione .NET e Java devono funzionare allo stesso modo, senza differenze. Tuttavia non è stato trascurato l’aspetto delle prestazioni, nel senso che l’obbiettivo è stato quello di fare in modo che la versione .NET dell’interprete possa funzionare con una certa velocità o comunque il linea con le prestazioni della versione Java.

Infine, poiché l’interprete tuProlog permette l’interoperabilità tra il linguaggio Prolog e Java si è deciso di realizzare, nella versione .NET, il supporto per l’interoperabilità tra Prolog, Java e i principali linguaggi per la piattaforma .NET (C#, F#, VB.NET). Rendendo in tal modo possibile la realizzazione di applicazioni multi – paradigma in grado di sfruttare le peculiarità e i punti di forza degli ambienti Prolog, Java e .NET, ottenendo anche la possibilità di utilizzare il linguaggio Prolog come ponte tra i mondi .NET e Java.

# IKVM.NET

IKVM.NET[1] è un’implementazione di Java[2] per la piattaforma .NET[3], ed in quanto tale è costituito da diversi componenti:

* Java Virtual Machine implementata in .NET;
* libreria di classi Java, basata su OpenJDK 6[4], implementata in .NET;
* strumenti per l’interoperabilità tra Java e .NET.

IKVM.NET permette di tradurre il codice Java, in particolare il bytecode[5], in Common Intermediate Language[6] (CIL) consentendo l’esecuzione dell’applicazione su piattaforme .NET.

IKVM.NET rappresenta quindi un vero e proprio strumento di interoperabilità tra il mondo Java e il mondo .NET dato che permette a sviluppatori .NET di utilizzare librerie scritte in Java senza nessuna modifica e al tempo stesso permette di utilizzare il linguaggio Java come linguaggio per scrivere applicazioni .NET.

IKVM.NET supporta sia la piattaforma .NET di Microsoft[7] che la piattaforma Open Source Mono[8].

Lo sviluppo di IKVM.NET è portato avanti da un piccolo gruppo di programmatori tra i quali Jeroen Frijters, creatore del progetto stesso.

IKVM.NET è distribuito con licenza “zlib”[9], Open Source e molto permissiva.

## Requisiti

Per il corretto funzionamento, IKVM.NET richiede la presenza di un ambiente di esecuzione .NET (CLR), quindi è necessario installare il Microsoft .NET Framework 2.0 (o superiore) oppure il Framework Mono 2.0 (o superiore).

IKVM.NET, pensato come ponte tra i mondi Java e .NET non dispone di un compilatore Java, quindi nel caso in cui si voglia scrivere applicazioni in Java da eseguire poi su piattaforma .NET sarà necessario dotarsi di un qualsiasi compilatore Java standard.

## Installazione

IKVM.NET è distribuito sia per macchine con architettura x86 che x64.

Per installare IKVM.NET è necessario soltanto estrarre i file contenuti nell’archivio scaricato.

Per scrivere applicazioni che utilizzano IKVM.NET è necessario inserire le librerie nella Global Assembly Cache (GAC) oppure nella directory dell’applicazione che si sta sviluppando, per permettere all’ambiente .NET di trovarle.

IKVM.NET è distribuito anche sotto forma di pacchetto contenente i file sorgente sia degli strumenti di interoperabilità che delle librerie che implementano la JVM e la libreria di classi standard.

## Modalità dinamica

Dato che IKVM.NET include una Java Virtual Machine implementata in .NET è in grado di eseguire qualunque applicazione Java direttamente sulla piattaforma .NET.

In questa modalità, le applicazioni Java vengono tradotte al volo in CIL e avviate nell’ambiente .NET.

Lo strumento che permette di sfruttare questa modalità è *ikvm.exe* che può essere inteso come corrispondente a *java.exe* su piattaforma Java. Infatti *ikvm.exe* svolge lo stesso compito di *java.exe*, avviare un’applicazione Java, con l’unica differenza che la piattaforma target è differente. Tale corrispondenza si evince anche dalle opzioni disponibili, molte delle quali sono comuni ad entrambi gli strumenti. Tra le opzioni, le più importanti sono sicuramente:

* cp <dir|jar|zip>: per specificare le directory o i file in cui cercare le diverse parti dell’applicazione;
* D<nome>=<valore>: per impostare una variabile di sistema.

Quindi è possibile utilizzare lo strumento *ikvm.exe* in modo quasi trasparente rispetto ad una qualsiasi JVM, ottenendo così il vantaggio di poter utilizzare un’applicazione Java senza disporre dell’ambiente di esecuzione.

In questa modalità è supportato appieno il modello di class loading di Java, quindi *ikvm.exe* sarà in grado di caricare tutte le classi che si trovano nel class path specificato come variabile d’ambiente del sistema operativo oppure quello specificato con l’opzione *–cp*, proprio come avviene in Java.

L’unico aspetto carente di tale modalità è quello riguardante le prestazioni, infatti ad ogni caricamento di un tipo, quest’ultimo, prima di essere utilizzato dovrà essere tradotto in CIL con ovvi sprechi di tempo.

## Modalità statica

Utilizzando questa modalità è possibile tradurre un’applicazione oppure una libreria Java in CIL e memorizzare il codice in un assembly tipico della piattaforma .NET (file con estensione *.exe* oppure *.dll*). A questo punto gli assembly generati possono essere inclusi in un’applicazione .NET, scritta in un qualunque linguaggio supportato, potendo così utilizzare le “classi Java” come se fossero classi .NET in modo del tutto trasparente. Di fatto i tipi presenti nell’assembly generato sono a tutti gli effetti tipi .NET creati a partire dai corrispondenti tipi Java, quindi non c’è nessuno strato intermedio.

La traduzione da Java bytecode a CIL è svolta dallo strumento *ikvmc.exe* che può corrispondere allo strumento *javac.exe* su piattaforma Java con la fondamentale differenza che *ikvmc.exe* traduce il codice che è già stato compilato, rendendo potenzialmente possibile utilizzare, in un’applicazione .NET, una qualunque libreria Java anche senza disporre dei relativi file sorgente.

Le opzioni disponibili per *ikvmc.exe* sono tipiche della piattaforma .NET. Infatti è possibile specificare le informazioni per firmare l’assembly, per indicare la piattaforma target (x86, x64) oppure per indicare quale formato avrà l’assembly generato (*.exe*/*.dll*). Nei paragrafi successivi verrà condotta una trattazione più completa delle opzioni disponibili.

In questa modalità non è pienamente supportato il modello di class loading di Java, come approfondito nei paragrafi successivi.

## Tradurre applicazioni Java in .NET

Nella seguente sezione viene approfondito l’utilizzo di IKVM.NET per tradurre applicazioni Java in .NET e per lo sviluppo di applicazioni .NET che utilizzano le librerie Java tradotte.

Come accennato in precedenza lo strumento fondamentale per queste operazioni è *ikvmc.exe*.

### Standard Class Library

IKVM.NET viene fornito con una implementazione in .NET della libreria standard di Java quindi è possibile utilizzare tutti i tipi Java presenti nella libreria in applicazioni .NET. L’unica differenza da tenere in considerazione è che i package Java sono convertiti in namespace, quindi ad esempio il namespace java.lang rappresenta l’omonimo package.

La libreria standard Java è stata suddivisa in diversi assembly ciascuno dei quali con nome *IKVM.OpenJDK.\*.dll*, dove l’asterisco indica un particolare sottoinsieme della intera libreria, ad esempio l’assembly *IKVM.OpenJDK.Media.dll* contiene tutti i tipi necessari per l’utilizzo di suoni o immagini. Nell’assembly *IKVM.OpenJDK.Core.dll* invece sono presenti tutti i package principali come ad esempio java.lang, java.io, java.util, java.awt, e altri. Per poter utilizzare quindi la libreria standard in un’applicazione .NET sarà sufficiente aggiungere un riferimento all’assembly IKVM necessario e fare in modo che l’ambiente .NET sia in grado di localizzarlo a tempo di esecuzione inserendolo nella GAC oppure nella cartella corrente dell’applicazione.

Tutte le API della libreria standard sono presenti ma alcune di queste semplicemente non funzionano perché non sono implementate, di seguito è riportata una lista dei package non implementati aggiornata alla versione 0.44.0.5.

|  |  |
| --- | --- |
| com.sun.security.auth.module | Non implementato. |
| java.applet | Non implementato. |
| java.awt | Implementazione parziale basata su System.Windows.Assembly |
| java.io.Console | Non implementato. |
| java.lang.instrument | Non implementato. |
| java.lang.management | Non implementato. |
| java.net | Non implementato il supporto a IPv6. |
| java.net.ProxySelector | L’ottenimento del proxy di sistema di default non è implementato. |
| java.util.zip | Implementazione parziale basata su GNU Classpath. |
| javax.imageio.plugins.jpeg | Implementazione parziale. Le immagini JPEG possono essere lette e scritte ma non c’è il supporto per i metadata. |
| javax.management | Non implementato. |
| javax.print | Non implementato. |
| javax.script | Non implementato. |
| javax.smartcardio | Non implementato. |
| javax.sound | Non implementato. |
| javax.tools | Non implementato. |
| org.ietfs.jgss | Non implementato. |
| sun.jdbc.odbc | Implementazione parziale basata su “.NET ODBC managed provider”. |
| sun.net.www.content.audio | Non sono implementati i gestori per i contenuti audio. |
| sun.net.www.content.image | Non sono implementati i gestori per le immagini. |

*Tabella 1 – Elenco package non implementati IKVM.NET*

Un’ulteriore limitazione della class library nella versione 0.44.0.5 riguarda i tipi Generici. Questa funzionalità non è ancora supportata da IKVM.NET, quindi ad esempio tutte le collezioni presenti nel package java.util non sono generiche.

### Librerie esterne

Nel caso in cui si desideri utilizzare una libreria (.jar) o anche una singola classe Java (.class) in un’applicazione .NET sarà necessario prima di tutto tradurre il bytecode in CIL. Durante questa fase si dovrà fare attenzione alle dipendenze della libreria Java, cioè se la libreria A.jar per il corretto funzionamento necessità di altre librerie, B.jar e C.jar, sarà necessario tradurre prima tali librerie e poi all’atto della traduzione della libreria A.jar aggiungere un riferimento agli assembly generati in precedenza (B.dll e C.dll).

Come accennato precedentemente *ikvmc.exe* mette a disposizione diverse opzioni per controllare il processo di traduzione, come ad esempio:

* assembly : <name>: specifica il nome dell’assembly generato;
* target : <exe|winexe|library|module>: specifica il tipo di assembly generato;
* platform : <x86|x64|Itanium|any>: specifica la piattaforma target;
* keyfile : <keyfilename>: specifica il file necessario per firmare l’assembly;
* version : <M.m.b.r>: specifica la versione dell’assembly;
* r : <file>: specifica un riferimento ad un altro assembly (utilizzata per le dipendenze);
* classloader <class>: specifica della classe che verrà utilizzata come class loader (vedere paragrafi successivi);
* debug: vengono create le informazioni necessarie per il debugging.

Lo strumento *ikvmc.exe* permette anche di tradurre diversi file di input (.class/.jar) in un unico assembly e nel caso in cui siano presenti classi o risorse duplicate, lo strumento utilizzerà solo la prima occorrenza ignorando le altre e producendo un messaggio di avviso.

È da tenere in considerazione che le classi o i metodi con visibilità di package in Java saranno tradotti in .NET con visibilità internal, quindi impossibili da utilizzare da altri assembly.

### Sistema di Class Loading

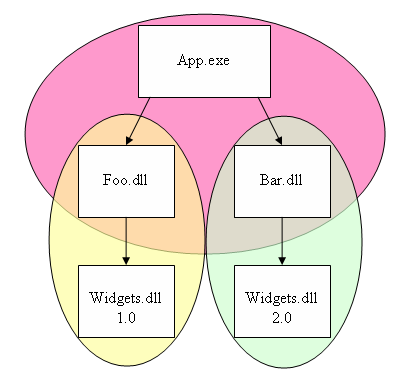
Java e .NET hanno un meccanismo di risoluzione dei componenti delle applicazioni decisamente diverso e soprattutto incompatibile. L’ambiente di esecuzione Java, infatti, nella ricerca delle classi si basa principalmente sul classpath, un elenco di cartelle o archivi in cui cercare la classe richiesta. Il concetto di classpath non esiste in .NET che invece utilizza delle modalità di ricerca differenti (cartella corrente, GAC, file di configurazione), questa discrepanza fa si che nel momento in cui un’applicazione scritta per l’ambiente Java viene tradotta in .NET, dove mancano diversi concetti su cui l’applicazione stessa si basa si originino alcuni problemi relativi al caricamento delle classi.

Di seguito sarà analizzato più in dettaglio il sistema adottato da IKVM.NET per caricare classi Java e .NET.

Ad ogni assembly generato staticamente viene associato un class loader, che può essere specificato mediante l’apposita opzione oppure a default è un’istanza della classe ikvm.runtime.AssemblyClassLoader.

Il class loader di default utilizza il seguente algoritmo per localizzare una classe o una risorsa:

1. controlla la presenza nell'assembly a cui è stato associato,
2. controlla la presenza in tutti gli assembly direttamente referenziati dall'assembly a cui è associato.



*Figura 1 – Class Loader modalità statica IKVM.NET*

Come mostrato in Figura 1 il class loader di *App.exe* (ellisse rosa) è in grado di trovare qualunque classe o risorsa presente negli assembly *Foo.dll* e *Bar.dll* ma non riesce a raggiungere quelle presenti negli assembly *Widgets.dll*, raggiungibili invece dai class loader di *Foo.dll* e *Bar.dll* (ellisse giallo e azzurro).

Il class loader di default si adatta bene ad applicazioni in cui tutte le dipendenze sono soddisfatte staticamente, cioè nel caso in cui l’applicazione definisca un riferimento ad alcune librerie e le utilizzi direttamente; perché durante la fase di traduzione sarà necessario aggiungere i riferimenti alle librerie e quindi l’AssemblyClassLoaderriuscirà a caricare le classi dato che il relativo assembly è direttamente referenziato. Nel caso in cui invece l’applicazione carichi dinamicamente le classi di cui ha bisogno, la versione tradotta in .NET non sarà in grado di caricare tali componenti, dato che non esiste il riferimento alla libreria.

Di seguito vengono elencate alcune possibili soluzioni che sono state analizzate per colmare le mancanze nel sistema di class loading dinamico di IKVM.NET.

* **Creare un unico assembly**

La soluzione più semplice è sicuramente tradurre sia l’applicazione che le librerie utilizzate in un unico assembly, in questo modo non si hanno problemi di caricamento delle classi, dato il class loader è unico.

Questa soluzione non è comunque priva di difetti, infatti in questo caso l’applicazione diventerà più pesante dal punto di vista della memoria occupata durante l’esecuzione dato che occorre caricare tutti i possibili componenti anche se poi se ne usa uno solo. Ad esempio occorrerebbe caricare tutti i driver JDBC supportati. Inoltre le risorse duplicate saranno perse.

* **Aggiungere un riferimento statico alla libreria**

Questa soluzione consiste nell'aggiungere un riferimento(opzione –r) alla libreria che deve essere caricata, quando si traduce l'applicazione che utilizza dinamicamente tale libreria.

In tal modo l'applicazione si comporterà, dal punto di vista del class loading, come una qualunque applicazione .NET a cui sia stato aggiunto un riferimento statico ad un assembly esterno, seguendo quindi gli algoritmi di ricerca standard della piattaforma .NET.

In tal modo si risolve il principale problema della soluzione precedente poiché non viene caricato niente fino a che non viene richiesto ma si perde l’utilità del caricamento dinamico. Infatti, dal momento in cui si aggiunge il riferimento, l’ambiente di esecuzione .NET cercherà di localizzare esattamente l’assembly referenziato (stessa versione) quindi l’aggiornamento di una libreria esterna comporterebbe anche la ritraduzione dell’applicazione principale per poter modificare il riferimento.

* **Utilizzare un AppDomainAssemblyClassLoader**

Questa soluzione prevede di utilizzare un class loader messo a disposizione da IKVM.NET, in particolare ikvm.runtime.AppDomainAssemblyClassLoader.

Tale loader è in grado di caricare classi che sono presenti in un qualunque assembly correntemente caricato nell'AppDomain dell'applicazione. Se la classe richiesta non viene trovata in un'assembly già caricato verrà effettuata la ricerca classica negli assembly direttamente referenziati (come nel class loader di default).

Questa soluzione risolve solo in parte il problema della memoria occupata durante l’esecuzione perché sarebbe necessario caricare tutti gli assembly nell’AppDomain prima ancora di utilizzarli effettivamente.

* **Utilizzare uno sharedclassloader**

Un’altra funzionalità messa a disposizione da IKVM.NET, prevede di utilizzare un class loader condiviso tra diverse librerie che è in grado di caricare le classi presenti in tutti gli assembly che lo condividono.

Per sfruttare questa funzionalità occorre invocare lo strumento *ikvmc.exe* in modo particolare:

ikvmc -sharedclassloader { first.jar } { second.jar } { third.jar }

Questa opzione però può essere utilizzata solo quando si traducono jar (o class) distinti in assembly distinti con un unico comando, con ovvi problemi si scrittura del comando stesso che potrebbe diventare anche molto complesso perché per ogni libreria occorre specificare delle opzioni particolari. Inoltre in caso di aggiornamento di una libreria sarebbe comunque necessario tradurre nuovamente l’intera applicazione per fare in modo che il class loader condiviso abbia un riferimento alla nuova versione della libreria.

* **Utilizzare un ClassPathAssemblyClassLoader**

Un ClassPathAssemblyClassLoaderè un altro class loader messo a disposizone da IKVM.NET. Tale loader effettua inizialmente una ricerca classica, con lo stesso algoritmo adottato da AssemblyClassLoader, dopodiché cerca la classe nel class path sotto forma di file .class/.jar.

Qundi questo class loader cerca le classi o le librerie Java e le traduce dinamicamente in .NET. L'applcazione si comporta a questo punto come una qualunque applicazione java che tenta di caricare le classi dal class path con la differenza che l’impostazione del class path può essere fatta soltanto modificando la corrispondente variabile d’ambiente del sistema operativo e non è possibile farlo con l'opzione –cp.

Questa soluzione presenta il significativo vantaggio di semplificare l’aggiornamento di una libreria, infatti è sufficiente sostituire il vecchio file con il nuovo (non è necessario neanche tradurlo) senza modificare o ritradurre l’applicazione principale. Tale vantaggio non è comunque assoluto: il processo di traduzione a tempo di esecuzione, infatti, non avviene in un tempo trascurabile, quindi in applicazioni che caricano dinamicamente componenti con una frequenza elevata si potrebbe avere un calo di prestazioni considerevole.

* **Scrivere il proprio loader**

È possibile scrivere il proprio class loader semplicemente derivando dalla classe ClassLoaderoppure riutilizzando altri loader, ad esempio derivando da URLClasseLoader. Questa è la soluzione più flessibile dato che è possibile scegliere l’algoritmo e le modalità di caricamento delle classi.

Un altro aspetto da tenere in considerazione per quanto riguarda il class loading è il fatto che nella implementazione della Java Library in .NET il comportamento del metodo statico Class.forName(String)è stato esteso per consentire il caricamento di classi contenute in assembly che non sono referenziati direttamente, specificando l’AssemblyQualifiedName della classe. Di seguito viene riportato un esempio di invocazione:

Class c = Class.forName("System.Windows.Forms.MessageBox, " +

"System.Windows.Forms, " +

"Version=2.0.0.0, " +

"Culture=neutral, " +

"PublicKeyToken=b77a5c561934e089");

Se non vengono specificate tutte le informazioni relative l’assembly, la ricerca viene effettuata solo nella cartella corrente dell’applicazione.

Questa possibilità fa si che nel caso in cui l’applicazione Java utilizzi per il caricamento dinamico il metodo Class.forName(String), dopo la traduzione in CIL è possibile utilizzare la stessa applicazione scritta originariamente in Java per caricare dinamicamente ed utilizzare classi scritte in .NET senza apportare modifiche al codice.

### Debugging

Il debugging delle applicazioni Java tradotte in .NET è possibile direttamente sul codice Java stesso, a patto di avere a disposizione i sorgenti. Durante la fase di traduzione, specificando l’opportuna opzione, verrà generato un file con estensione .pdb (Program Debug Database) in cui vengono memorizzate tutte le informazioni necessarie per il debugging. Utilizzando Microsoft Visual Studio, è sufficiente inserire il file *.pdb* nella stessa directory in cui si trova l’applicazione da debuggare, a questo punto l’IDE richiede i file sorgenti dell’applicazione e durante l’esecuzione è possibile vedere l’avanzamento direttamente sul codice Java. Con questa tecnica di debugging è anche possibile inserire breakpoints all’interno del codice e sfruttare l’inspection delle variabili, il tutto comunque sul codice Java dalla quale è stata tradotta l’applicazione.

## Applicazioni .NET in Java

Nella seguente sezione vengono analizzati gli strumenti messi a disposizione da IKVM.NET per scrivere applicazioni .NET utilizzando il linguaggio Java e come vengono colmate le differenze linguistiche tra Java e CIL.

### Strumento *ikvmstub.exe*

Il principale strumento per scrivere applicazioni .NET in Java è *ikvmstub.exe* che si occupa di generare un file con estensione .jar a partire da un file assembly (estensione *.dll* o *.exe*). Il file generato contiene tutte le classi e le interfacce Java corrispondenti a quelle .NET ma non contiene alcun codice di implementazione, contiene solo il minimo indispensabile per soddisfare il compilatore Java e per permettergli di effettuare il controllo dei tipi. In questo modo il file generato può essere incluso come libreria esterna in un qualsiasi IDE Java e sfruttare quindi il code completion dell'IDE stesso, ovviamente non si può sfruttare il debugger. A questo punto è possibile scrivere un’applicazione in linguaggio Java che utilizza le classi offerte dal Framework .NET. L’applicazione risultante dovrà essere compilata con un qualsiasi compilatore Java, indicando nel classpath proprio il jar generato in precedenza, ovviamente l'applicazione non potrà essere eseguita su una JVM standard dato che il jar non contiene implementazione, ma dovrà essere tradotta con *ikvmc.exe* ed eseguita nell'ambiente .NET.

In questo modo è possibile creare un'applicazione .NET a tutti gli effetti ma scritta in linguaggio Java potendo sfruttare l’auto-completamento del codice messo a disposizione dagli IDE e soprattutto il controllo dei tipi implementato dal compilatore Java.

### Caratteristiche .NET in Java

Quando lo strumento *ikvmstub.exe* genera il file jar deve evitare conflitti sui nomi tra le classi Java e quelle .NET e soprattutto deve incapsulare le caratteristiche offerte dal linguaggio CIL, come ad esempio le proprietà, i delegati, gli eventi, nei costrutti messi a disposizione dal linguaggio Java. Quindi deve realizzare una sorta di conversione che permetta di utilizzare tutta l’espressività messa a disposizione dei linguaggi .NET utilizzando però il linguaggio Java.

Di seguito è riportato un elenco delle conversioni adottate da *ikvmstub.exe*.

* **Namespace**

Sono convertiti in package con prefisso “cli.” per prevenire possibile conflitti sui nomi.

* **Proprietà**

Sono convertite in metodi con prefisso “get\_” e “set\_”.

* **Enumerativi**

Sono convertiti in classi che derivano da cli.System.Enum e prevedono campi statici e interi corrispondenti ai diversi valori dell’enumerativo stesso.

* **Delegati**

Sono convertiti in una classe e un’interfaccia.

La classe ha lo stesso nome del delegato .NET e deriva da System.MulticastDelegate*.*

L'interfaccia è annidata all'interno della classe ed ha sempre il nome Method e definisce un metodo Invoke(...) che rappresenta la firma del delegato. Questo è il metodo che viene invocato quando viene scatenato l'evento.

La classe prevede di essere costruita passandogli un riferimento ad un oggetto che implementi l'interfaccia Method.

* **Eventi**

Sono convertiti in metodi con prefisso “add\_” e “remove\_”. Tali metodi accettano come parametro un oggetto della classe che rappresenta il delegato.

* **Params**

Viene convertito in un array di Object.

* **Attributi**

Vengono tradotti in una classe con lo stesso nome dell’attributo e con i metodi “get\_” e “set\_” corrispondenti alle proprietà definite nell’attributo.

Annidata nella classe c'è un'Annotazione Java di nome Annotationche definisce i metodi con lo stesso nome delle proprietà definite nell'attributo. Tali metodi possono essere comunque utilizzati solo per ottenere i valori dell’annotazione ma non per modificarli (metodi solo in lettura) anche se nell’attributo .NET le proprietà sono in scrittura.

Utilizzando il linguaggio Java quindi se si vuole annotare una parte di codice si deve utilizzare l’annotazione annidata nella classe, comunque una volta tradotta l’applicazione in CIL l’annotazione viene tradotta nel corrispondente attributo.

Nell’appendice B è riportato un esempio di applicazione .NET scritta in Java.

### Debugging

Dopo avere tradotto l’applicazione Java in .NET le modalità di debugging sono le stesse descritte nella sezione 1.5.4.

# Ja.NET

Ja.NET[10] è un’implementazione di Java per la piattaforma .NET ed è costituito da:

1. Java Virtual Machine implementata in .NET;
2. libreria di classi standard Java, basata su Apache Harmony[11], implementata in .NET;
3. strumenti per la compilazione e l’esecuzione di applicazioni Java su piattaforma .NET.

Ja.NET offre anche tutti gli strumenti necessari per lo sviluppo di applicazioni su piattaforma Java standard, quindi fornisce anche una JVM standard, una libreria di classi standard e gli strumenti standard per Java.

Ja.NET è in grado di compilare un’applicazione scritta in linguaggio Java in una che può essere eseguita sulla piattaforma .NET, quindi compila direttamente il codice sorgente in CIL.

Ja.NET rende possibile utilizzare librerie scritte in Java in applicazioni .NET e scrivere applicazioni .NET utilizzando il linguaggio Java. Quindi prevede delle possibilità di utilizzo del tutto analoghe a quelle offerte da IKVM.NET, comunque con modalità leggermente differenti.

Ja.NET è un progetto open source distribuito con la licenza “Eclipse Public License - v 1.0”[12].

## Requisiti

Per poter utilizzare Ja.NET occorre avere installato sulla macchina target il Microsoft .NET Framework 3.5 (o superiore) e le Visual C++ 2008 runtime libraries[13].

Ja.NET non supporta Mono e quindi è in grado di funzionare solo su Microsoft Windows.

## Installazione

Ja.NET è distribuito solo per macchine con architettura x86.

È possibile installare Ja.NET in due forme:

* **Ja.NET SE Runtime Environment**: contenente tutti i component necessari per eseguire applicazioni Java sia sulla JVM standard che sulla piattaforma .NET;
* **Ja.NET SE Development Kit**: contenente sia l’ambiente di esecuzione che tutti gli strumenti necessari per sviluppare applicazioni sia per l’ambiente Java che per la piattaforma .NET.

Entrambe le distribuzioni possono essere scaricate sotto forma di pacchetto autoinstallante per Windows (*.msi*) oppure come archivio (*.zip*) contenente i sorgenti.

Nel caso in cui si scelga il pacchetto autoinstallante, il sistema sarà configurato automaticamente, altrimenti è necessario inserire le librerie di Ja.NET nella GAC oppure nella cartella dell’applicazione che si sta sviluppando, per permettere all’ambiente .NET di localizzarle durante l’esecuzione.

**Nota:** dato che gli strumenti di Ja.NET hanno lo stesso nome degli strumenti standard di Java (esempio *java.exe*, *javac.exe*, ecc.) se si installa Ja.NET su una macchina dove è già stato installato un Java Development Kit standard (es. Oracle) è conveniente realizzare degli script *.bat*, con nomi diversi, che eseguano i relativi strumenti di Ja.NET. Ad esempio è possibile realizzare lo script *ja.bat* che esegue *java.exe* di Ja.NET. Questo accorgimento risolve il problema dell’omonimia dei comandi senza intervenire sulle variabili d’ambiente del sistema operativo.

## Modalità statica

In questa modalità Ja.NET compila i file sorgente Java in particolari file con estensione .class che sono comunque assembly .NET a tutti gli effetti e possono quindi essere utilizzati in qualsiasi applicazione .NET. Questa scelta è dettata dal fatto che gli sviluppatori di Ja.NET volevano rendere il processo di sviluppo di applicazioni .NET quanto più simile possibile allo sviluppo di applicazioni sulla piattaforma Java standard. Per lo stesso motivo i principali comandi di hanno lo stesso nome dei comandi della piattaforma standard. È comunque possibile mediante una particolare opzione generare un file assembly classico, con estensione *.dll*.

Lo strumento da utilizzare per la compilazione è *javac.exe*.Tale strumento presenta un insieme di opzioni del tutto identiche a quelle disponibili sul compilatore per la piattaforma Java standard e altre invece specifiche per Ja.NET. Una trattazione più approfondita dello strumento *javac.exe* sarà condotta nei paragrafi successivi.

## Compilare applicazioni Java in .NET

### Standard Class Library

Ja.NET fornisce un’implementazione della libreria standard di Java in .NET, quindi è possibile utilizzare tutte le classi della libreria standard in applicazioni .NET. L’unica differenza sta nel fatto che ogni package disponibile nella libreria standard è convertito in un namespace avente lo stesso nome del package. Ad esempio il package java.iopuò essere utilizzando importando il namespace java.io.

L’intera libreria è stata suddivisa in diversi assembly, ciascuno dei quali contiene l’implementazione di una parte della libreria stessa. Il nome degli assembly segue lo schema seguente: *janet.\*.dll* dove l’asterisco identifica la parte della libreria contenuta. Ad esempio l’assembly *janet.rmi.dll* contiene tutti i package necessari per l’invocazione di metodi remoti. All’interno dell’assembly *janet.luni.dll* sono invece contenuti i package più comunemente utilizzati in un’applicazione Java, quali: java.lang, java.io, java.net e java.util.

Per l’attuale versione di Ja.NET (1.5.0.365) non esiste una lista dei package non implementati. Dallo studio è emerso che entrambi i package grafici java.awte java.swingnon sono implementati, quindi le applicazioni Java che utilizzano questi package vengono compilate correttamente in CIL ma al momento dell’esecuzione viene lanciata un eccezione. Altri package non implementati sono org.omg.CORBA e com.sun.jdi.

I tipi generici di Java non sono supportati, quindi ad esempio tutti i metodi che nella implementazione standard accettano un tipo generico, nella versione .NET accettano semplicemente un riferimento a Object.

### Librerie esterne

Come accennato in precedenza, utilizzando lo strumento *javac.exe* è possibile compilare i sorgenti di una libreria Java in CIL per poterla utilizzare all’interno di applicazioni .NET. L’utilizzo di tale strumento è del tutto equivalente all’utilizzo dell’omonimo strumentosulla piattaforma Java standard. Infatti, durante la compilazione occorre specificare tutti i file con estensione *.java* che devono essere compilati e indicare il class path in cui il compilatore si aspetta di trovare le classi referenziate all’interno dei sorgenti. Il concetto di class path è stato inoltre esteso per supportare anche gli assembly, quindi è possibile inserire nel class path cartelle, file jar e assembly .NET (*.dll/.exe*). In questo modo è possibile utilizzare il linguaggio Java per scrivere applicazioni che utilizzano delle librerie .NET, come specificato nella sezione 2.5.

Lo stesso strumento *javac.exe* può essere utilizzato per compilare i sorgenti in bytecode standard per poter utilizzare la libreria sulla piattaforma Java standard. Questa intenzione deve essere esplicitata mediante una particolare opzione del compilatore.

Tra tutte le opzioni disponibili per il compilatore, quelle specifiche per Ja.NET sono:

* java: indica che i sorgenti devono essere compilati per la piattaforma Java standard. Se tale opzione non viene indicata, a default, il compilatore produce codice in CIL;
* both: indica che il compilatore deve produrre sia i file compilati in bytecode standard che in CIL;
* secureScope: specifica che i componenti dell’applicazione (classi, metodi) con visibilità di package in Java siano compilati con visibilità internalin .NET. Se non è indicata questa opzione i componenti con visibilità di package saranno compilati con visibilità pubblica;
* bam : <assembly-name>: specifica il nome che avrà il file in uscita dal processo di compilazione. Indicando questa opzione il file generato sarà un assembly con estensione *.dll*. Se non specificata sarà prodotto un assembly con estensione *.class*;
* g: vengono create le informazioni per il debugging.

Per controllare in modo più accurato il processo di produzione di un assembly, Ja.NET mette a disposizione un’ulteriore strumento, *bam.exe* (*Build Assembly Module*). Tale applicazione è in grado di fondere diversi assembly (generati con *javac.exe* oppure con strumenti standard della piattaforma .NET) in un unico assembly .NET con estensione *.dll*.

Le opzioni disponibili per *bam.exe* sono tipiche della piattaforma .NET; le più significative sono:

* k: specifica il file necessario per firmare l’assembly, per poterlo inserire nella GAC;
* r: specifica la lista degli assembly da utilizzare per risolvere le dipendenze;
* m: specifica il file manifest da utilizzare per la generazione dell’assembly;
* g: vengono create le informazioni per il debugging.

### Esecuzione

Ja.NET mette a disposizione un strumento (*java.exe*) che consente di eseguire applicazioni sia sulla JVM Harmony che sulla piattaforma .NET. In quest’ultimo caso ovviamente l’applicazione deve essere compilata con *javac.exe*. Questo strumento presenta un funzionamento del tutto equivalente all’omonimo strumento sulla piattaforma Java standard con la differenza che a default, lo strumento Ja.NET, esegue l’applicazione sulla piattaforma .NET. Come per *javac.exe* il concetto di class path è stato esteso per supportare gli assembly.

Per eseguire un’applicazione sulla JVM Harmony occorre specificare l’opzione –vmdir:drlvm.

Come approfondito nella sezione successiva, utilizzando il launcher *java.exe* il sistema di class loading di Java è pienamente supportato, mentre includendo una libreria in un’applicazione .NET si hanno alcuni problemi riguardanti il caricamento delle classi.

### Sistema di Class Loading

Come evidenziato nella sezione 1.5.3, la piattaforma Java e .NET hanno un sistema di localizzazione delle classi differente e incompatibile. La differenza principale è costituita dalle diverse posizioni in cui sono cercate le classi: in Java nel class path, in .NET nella cartella corrente oppure nella GAC (principalmente). Tale differenza è mascherata dal launcher integrato in Ja.NET (*java.exe*). Questo è reso possibile dall’introduzione del concetto di class path anche per l’esecuzione di applicazioni .NET. In tal modo, infatti, il launcher esegue l’applicazione sulla piattaforma .NET e predispone un sistema di class loading che è in grado di caricare le classi dagli assembly referenziati nel class path, simulando di fatto il sistema tipico di Java.

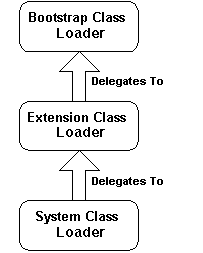
Di seguito sarà analizzato più dettagliatamente il sistema di class loading predisposto da *java.exe*.

Prima dell’esecuzione vera e propria viene creata una gerarchia di class loaders del tutto simile a quella presente in una JVM standard dove in più ogni loader mantiene il riferimento agli assembly caricati (ogni assembly in ogni momento può essere associato ad un solo loader). Questa associazione viene mantenuta per fare in modo che tutte le classi presenti all’interno degli assembly referenziati siano visibili dal relativo class loader.

La gerarchia contiene inizialmente:

* **bootstrap class loader**: associato a tutti gli assembly attualmente caricati nel processo. Rappresenta la radice della gerarchia;
* **un extension class loader**: associato a tutti gli assembly presenti nella directory JANET\_INSTAL\jre\lib\ext;
* **system class loader**: associato a tutti gli assembly indicati nel class path. A default il class path è impostato sulla cartella corrente.

È anche possibile aggiungere un proprio class loader durante l’esecuzione.



*Figura 2 – Tipica gerarchia di class loaders Ja.NET*

L’algoritmo utilizzato per localizzare una classe è lo stesso adottato sulla piattaforma Java standard con l’unica differenza che la ricerca viene effettuata negli assembly associati a ciascun class loader.

L’algoritmo nel dettaglio prevede i seguenti passi:

1. il class loader interessato verifica che la classe non sia già stata caricata. Questa fase è svolta controllando la presenza della classe in tutti gli assembly associati al class loader. Se la classe viene trovata, il class loader la restituisce altrimenti si passa alla fase successiva;
2. nel caso in cui il class loader abbia un class loader genitore, cioè più in alto nella gerarchia, la risoluzione della classe è delegata a quest’ultimo;
3. se, dopo la delega, la classe non è stata trovata il class loader adotta la propria politica di ricerca. Ad esempio il System Class loader cercherà la classe in tutti gli assembly presenti nel class path.

L’assembly contenente la classe viene associato al primo class loader che localizza la classe stessa.

Nel caso in cui invece la classe sia caricata dalla piattaforma .NET, Ja.NET si deve occupare di associare l’assembly contenente la classe ad un class loader, per rendere visibili le classi che contiene. Questo caso si verifica quando, ad esempio, si utilizzano i metodi messi a disposizione dal framework .NET per caricare esplicitamente una classe o un assembly.

L'algoritmo utilizzato per decidere a quale class loader associare l’assembly prevede tre fasi:

* se l'assembly è caricato dalla GAC viene associato al bootstrap class loader;
* se l'assembly è caricato dalla directory corrente dell'applicazione o da una sub-directory viene associato al system class loader;
* altrimenti, viene determinata la classe che ha richiesto il caricamento e l’assembly viene associato al class loader di quest’utlima.

Quindi utilizzando il launcher fornito con Ja.NET non si riscontrano problemi relativi il class loanding.

Nel caso in cui invece, l’applicazione .NET che utilizza una libreria Java compilata in CIL, venga eseguita senza il launcher *java.exe*, si potrebbero avere dei problemi. Tali malfunzionamenti si manifestano soltanto nel caso in cui la libreria carichi dinamicamente classi con il metodo Class.forName(String) perché l’ambiente .NET, senza il supporto generato prima dell’esecuzione, dallo strumento *java.exe*, non è in grado di localizzare la classe.

Le possibili soluzioni al problema sono due:

* **Creare un unico assembly**

Questa soluzione consiste nel fondere l’applicazione e la libreria in un unico assembly utilizzando lo strumento *bam.exe*.

Questa soluzione, come descritto nel capitolo su IKVM.NET, porta a degli svantaggi dal punto di vista della memoria occupata durante l’esecuzione dato che occorre caricare tutti i possibili componenti anche se poi se ne usa uno solo.

* **Scrivere il proprio loader**

Questa soluzione prevede la realizzazione del proprio class loader che implementi le politiche di ricerca e localizzazione delle classi desiderate. È possibile scrivere il proprio class loader semplicemente derivando dalla classe ClassLoaderoppure da URLClasseLoader.

### Debugging

Il debugging avviene secondo le stesse modalità descritte nella sezione 1.5.4.

## Applicazioni .NET in Java

Ja.NET permette la realizzazione di applicazioni .NET utilizzando il linguaggio Java. A differenza di quanto avviene per IKVM.NET però, non esiste uno strumento che generi gli stub delle classi .NET da utilizzare. Questa mancanza porta a non poter sfruttare il code-completion offerto dagli IDE.

Una volta realizzata l’applicazione .NET in linguaggio Java, questa non potrà ovviamente essere eseguita su una JVM standard ma dovrà essere compilata con *javac.exe* per generare la versione .NET.

### Caratteristiche .NET in java

Nel CIL esistono dei costrutti linguistici dei quali non è presente la corrispondete versione in Java, ad esempio le proprietà o gli eventi. Quindi quando si scrivono applicazioni .NET in linguaggio Java è necessario adattare l’utilizzo di questi costrutti alle possibilità espressive offerte da Java.

Di seguito è riportato un elenco degli adattamenti nell’utilizzo delle caratteristiche del CIL:

* **Namespace**

Devono essere importati come si fa con i package.

* **Proprietà**

Sono utilizzate invocando dei metodi con prefisso “get\_” oppure “set\_” in base all’operazione che si vuole effettuare.

* **Enumerativi**

Sono utilizzati come in .NET.

* **Delegati / Eventi**

Sono convertiti in metodi con prefisso “add\_” e “remove\_”. Tali metodi accettano come parametro un oggetto della classe che rappresenta il delegato.

* **Params**

Un metodo che accetta come argomento un paramsdeve essere invocato passandogli un array di Object.

* **Attributi**

Sono utilizzati come se fossero delle annotazioni Java.

Nell’appendice D è riportato un esempio di applicazione .NET scritta in Java.

### Debugging

Dopo la compilazione in CIL le modalità di debugging sono le stesse indicate nella sezione 1.5.4.

# Confronto tra IKVM.NET e Ja.NET

In questo capitolo si mettono a confronto i due approcci per il porting di applicazioni Java su piattaforma .NET, messi a disposizione da IKVM.NET e Ja.NET con l’intento di capire quale dei due si adatti meglio a supportare un processo che comporti il minor costo possibile e sia sostenibile nel tempo.

## Usabilità

### Conversione da Java a .NET

Per quanto riguarda la possibilità di tradurre un’applicazione o una libreria Java in CIL, l’approccio di IKVM.NET risulta migliore. Il primo strumento analizzato, infatti, prevede la possibilità di tradurre l’applicazione già compilata, processando quindi il bytecode. Questa funzionalità non è da trascurare perché in applicazioni di una certa complessità è molto probabile che non si disponga di tutti i sorgenti, ad esempio quando si utilizzano librerie di terze parti non open source. In uno scenario del genere IKVM.NET consente comunque l’utilizzo di tali librerie sulla piattaforma .NET. Ja.NET invece, è in grado di tradurre un’applicazione Java in .NET solo se si dispone dei relativi sorgenti, limitando notevolmente le possibilità di utilizzo.

Un altro punto a favore di IKVM.NET è l’immediatezza nell’utilizzo. Con un solo comando è, infatti, possibile tradurre un’applicazione Java nella corrispondente versione .NET secondo il formato standard delle applicazioni eseguibili (file *.exe*). Lo strumento *ikvmc.exe* determina automaticamente la classe che rappresenta l’entry point dell’applicazione; questo è possibile dall’analisi del file MANIFEST all’interno del file jar dell’applicazione. Per quanto riguarda invece Ja.NET occorre prima compilare i sorgenti e poi realizzare l’eseguibile specificando il file MANIFEST opportuno.

### Applicazioni .NET in Java

Nel caso in cui si voglia scrivere un’applicazione per la piattaforma .NET utilizzando il linguaggio Java l’utilizzo di IKVM.NET è da preferire rispetto all’approccio offerto da Ja.NET. Questa affermazione è motivata soprattutto dal fatto che IKVM.NET, generando gli stub Java delle classi .NET (sezione 1.6.1), permette l’utilizzo del code-completion degli strumenti di sviluppo. Quest’ultimo è un fattore decisivo dato che le convenzioni utilizzate per l’utilizzo delle caratteristiche del CIL non sono proprie né del linguaggio Java né dei principali linguaggi .NET (paragrafi 1.6.2 e 2.5.1), quindi l’aiuto offerto dagli IDE è in questo caso indispensabile.

Ja.NET non fornisce una analoga funzionalità di IKVM.NET per la generazione degli stub delle classi .NET quindi non consente l’utilizzo del code-completion. Tuttavia permette in certi casi una maggiore trasparenza nell’utilizzo delle caratteristiche .NET mediante il linguaggio Java rispetto a quanto non faccia IKVM.NET. Un esempio di tale semplificazione si ha nella gestione degli attributi .NET che possono essere utilizzati come una qualunque annotazione standard. Inoltre Ja.NET permette lo sviluppo di applicazioni .NET con linguaggio Java senza la necessità di dotarsi di un compilatore Java standard, perché il sorgente dell’applicazione viene direttamente compilato in CIL da *javac.exe*, mentre con IKVM.NET occorre prima compilarlo con un compilatore standard e poi tradurlo con *ikvmc.exe*.

## Completezza

Come discusso nella sezione 2.4.1, il fatto che Ja.NET non implementi alcuni package molto importanti come java.swingoppure org.omg.CORBA limita fortemente l’utilizzo di applicazioni Java sulla piattaforma .NET. I componenti dell’applicazione che utilizzano i package non implementati dovrebbero, infatti, essere riscritti utilizzando gli strumenti equivalenti messi a disposizione dal framework .NET.

I package non implementati da IKVM.NET sono invece più marginali (gestione di smartcard, gestione di suoni, gestione degli script Java), quindi in applicazioni di una certa complessità è probabile che non vengano utilizzati, pertanto è molto difficile incontrare i problemi descritti per Ja.NET.

## Class Loading

Oltre ai problemi, già sottolineati nella sezione precedente, derivanti dalla mancanza di alcuni package nella libreria standard, altre limitazioni riguardano soprattutto il caricamento delle classi durante l’esecuzione. Anche per questo aspetto IKVM.NET risulta migliore rispetto alla contro parte perché offre alcune soluzioni aggiuntive (diverse tipologie di class loader disponibili) che si adattano a diverse esigenze.

Ja.NET d’altra parte permette di eseguire un’applicazione .NET come se fosse un’applicazione Java, in modo del tutto trasparente e senza problemi di class loading. Questa possibilità è però disponibile solo utilizzando il lancher java.exe. Nel caso in cui non si utilizzi tale strumento occorre tenere in considerazione i possibili problemi analizzati nella sezione 2.4.4 e optare per una delle due soluzioni possibili.

## Deployment

Un altro aspetto da tenere in considerazione è il numero di sistemi operativi sui quali può funzionare l’applicazione convertita in .NET. In questo caso risulta migliore l’approccio di IKVM.NET perché garantisce il funzionamento anche sulla piattaforma Mono e quindi anche su sistemi operativi come Linux e Mac OS. Ja.NET invece è funzionante solo su Windows.

## Supporto

IKVM.NET è seguito da una comunità di sviluppatori non indifferente e disponibile nel fornire supporto in caso di problemi. Non è difficile trovare in rete articoli che spiegano in dettaglio l’utilizzo o le funzionalità. Inoltre è stato osservato che circa ogni sei mesi viene rilasciata una nuova versione, con miglioramenti consistenti.

Per quanto riguarda Ja.NET invece la documentazione è abbastanza scarsa ed è quasi impossibile trovare ulteriori risorse in rete.

## Conclusione

I principali aspetti da tenere in considerazione quando si vuole realizzare un processo di porting automatico sono tre:

* **tempo**: deve essere possibile realizzare la conversione nel minor tempo possibile, per riuscire a fornire la nuova versione dell’applicazione in contemporanea su diverse piattaforme con il vantaggio però, di svilupparla per una sola;
* **costo**: deve essere possibile minimizzare le modifiche al codice esistente oppure la scrittura di componenti che non possono essere convertiti sulla piattaforma di destinazione, per evitare di vanificare l’obbiettivo del porting automatico;
* **sostenibilità nel tempo**: gli strumenti utilizzati per il processo devono garantire una certa longevità per evitare che, dopo qualche anno, sia impossibile effettuare la conversione perché lo strumento è diventato obsoleto.

IKVM.NET permette di ottenere ottimi risultati in tutti e tre gli aspetti, infatti, è possibile realizzare la versione .NET di un’applicazione in un tempo molto breve, con un solo comando. Inoltre è rara la possibilità di incontrare componenti non implementati che necessitano di essere scritti direttamente sulla piattaforma di destinazione. IKVM.NET rappresenta quindi un progetto già abbastanza maturo, e dato il numero di sostenitori in tutto il mondo, destinato a crescere ancora.

# Caso di studio: tuProlog

In questo capitolo dopo una breve analisi della struttura e delle principali funzioni messe a disposizione da tuProlog[14] viene effettuata la conversione di quest’ultimo verso la piattaforma .NET utilizzando lo strumento IKVM.NET.

Nonostante sia stato scelto IKVM.NET come strumento per effettuare il porting di tuProlog, si è deciso di analizzare anche la conversione basata su Ja.NET per sottolineare come quest’ultimo non sia adatto a supportare la conversione di applicazioni di complessità elevata.

Nel corso del capitolo sono state testate due versioni di tuProlog: la 2.1.1, attualmente la versione stabile, e la 2.3.0alpha, prossima versione di imminente rilascio.

## Introduzione

tuProlog è un interprete Prolog, sviluppato in Java presso il Dipartimento di Elettronica Informatica e Sistemistica della Facoltà di Ingegneria dell’Università degli Studi di Bologna. Le caratteristiche principali dell’interprete sono la leggerezza e la configurabilità[15].

La configurabilità è ottenuta grazie ad un meccanismo che consente di caricare e scaricare all'interno di un motore tuProlog predicati, funtori e operatori, grazie al concetto di libreria. Alcune librerie sono incluse nella distribuzione standard di tuProlog, altre possono essere scritte dall'utente.

tuProlog è fortemente integrato con Java, tanto da permettere l’utilizzo di oggetti Java all’interno di un’applicazione Prolog oppure l’utilizzo di un motore Prolog all’interno di applicazioni Java.

tuProlog è un progetto open source rilasciato sotto licenza “LGPL”[16].

## Librerie

Le librerie sono il mezzo attraverso il quale tuProlog realizza la sua caratteristica di configurabilità dinamica. Il motore è un nucleo minimale e leggero per scelta progettuale: include solo pochi predicati “built-in”, definiti staticamente al suo interno. Ogni altra forma di funzionalità, come predicati, funtori, flag ed operatori, viene fornita dalle librerie, le quali possono essere aggiunte o rimosse dal motore dinamicamente. In particolare, tuProlog include all'interno della sua distribuzione delle librerie standard, che sono caricate automaticamente nel motore. Queste librerie forniscono al motore Prolog le caratteristiche e le funzionalità di base, come predicati e funtori compatibili con gli standard ISO, predicati per l'I/O e per l'interoperabilità e l'integrazione tra l'ambiente Java e Prolog.

Le librerie standard sono:

* **BasicLibrary**: fornisce i predicati, i funtori e gli operatori di base di Prolog;
* **DCGLibrary**: fornisce il supporto per la “Definite Clause Grammar”, un'estensione delle grammatiche “context free” usata per descrivere linguaggi naturali e formali;
* **IOLibrary:** fornisce alcuni predicati di base per l'I/O;
* **ISOLibrary**: fornisce predicati e funtori che fanno parte della sezione built-in nello standard ISO e che non sono forniti dalle precedenti librerie;
* **JavaLibrary**: fornisce predicati e funtori che consentono di creare, accedere ed utilizzare le risorse Java (come classi ed oggetti).

## Interoperabilità con Java

L'integrazione con Java è completa e trasparente[17]: utilizzando il linguaggio Prolog è possibile creare, ed utilizzare ogni entità Java, grazie alla libreria JavaLibrary. Utilizzando il linguaggio Java, è dualmente possibile invocare ed usare un motore tuProlog come un qualsiasi oggetto. Quindi, per esempio, è possibile da un lato utilizzare direttamente da Prolog qualunque package Java come Swing, arricchendo, in questa maniera, l'interprete tuProlog della capacità di interagire con componenti grafici; dall'altro lato è possibile sfruttare dall'ambiente Java più motori tuProlog, eventualmente configurati in maniera diversa, ognuno con le proprie librerie.

Con l’ultima versione di tuProlog (2.3.0alpha) l’interoperabilità con Java è stata ulteriormente arricchita. Infatti, con l’introduzione del nuovo framework P@J[18], basato su annotazioni, è possibile specificare codice Prolog come implementazione di un metodo Java, senza la necessità di gestire in modo esplicito qualsiasi interazione tra i linguaggi. Ciò permette quindi in un certo senso di iniettare codice Prolog all’interno di un metodo Java, con il conseguente vantaggio di poter utilizzare un linguaggio logico, quale è Prolog, all’interno di un linguaggio Object Oriented in modo trasparente. Prima dell’introduzione di questo framework, per ottenere lo stesso comportamento sarebbe stato necessario istanziare un motore tuProlog configurato con l’opportuna teoria e invocare la soluzione di un obbiettivo, il tutto all’interno di un metodo.

## tuProlog su piattaforma .NET

tuProlog è sviluppato per la piattaforma Java. Esiste anche un’implementazione per la piattaforma .NET realizzata, nel corso degli anni, come lavoro di tesi di studenti della Facoltà di Ingegneria dell’Università degli Studi di Bologna. L’ultima tesi a riguardo è quella di Marco Albertin, con l’obbiettivo di garantire l’interoperabilità tra Prolog e diversi linguaggi .NET[19]. Questa versione non viene comunque mantenuta dal team di sviluppo principale di tuProlog, quindi la release attualmente disponibile per la piattaforma .NET non è in linea con la versione Java.

## Obbiettivo

Il primo obbiettivo è rendere possibile la realizzazione della versione per la piattaforma .NET dell’interprete, in modo del tutto automatico per fare in modo che, nel momento in cui sia disponibile la nuova versione per la piattaforma Java lo sia anche per quella .NET.

Il secondo obbiettivo, non meno importante, è assicurarsi che la versione dell’interprete per la piattaforma .NET si comporti esattamente allo stesso modo della versione Java.

Parte del lavoro di questa tesi è proprio quello di capire se i due progetti analizzati permettono di effettuare il porting automatico dell’interprete tuProlog.

## Piano di verifica

Per verificare quale dei due approcci risulti migliore per il porting dell’interprete tuProlog su piattaforma .NET si è operato secondo questo schema:

1. conversione dell’applicazione Java con gli strumenti messi a disposizione dai due progetti;
2. esecuzione degli stessi test previsti per la versione Java;
3. confronto dei risultati dei test tra la versione Java e la versione per la piattaforma .NET.

L’interprete è stato sottoposto a tre tipi di test:

* test di unità;
* test di accettazione;
* test delle prestazioni.

Tutti i test sono stati effettuati su una macchina con processore Intel Core 2 Duo T7300 – 2,00 GHz, con 2 GB di memoria RAM.

### Test di unità

L'obiettivo primario dei test di unità consiste nell'isolare la parte più piccola di software testabile nell'applicazione dal resto del codice per stabilire se funziona esattamente come previsto. Ogni singola unità viene sottoposta a test prima di essere successivamente integrata in moduli più complessi.

I test di unità per tuProlog prevedono il controllo del corretto funzionamento delle strutture dati di base dell’interprete, ossia le classi che derivano da Term, come Struct, Var e quelle numeriche (e.g. Int e Double), dei principali elementi del motore Prolog e di conseguenza del Parser, del gestore delle teorie, delle classi StructEnumerator e TermEnumeratore della principali librerie. Poiché la versione 2.3.0alpha dell’interprete presenta una gestione delle eccezioni migliorata, sono disponibili test aggiuntivi che ne controllano la correttezza.

Per effettuare i test di unità sulla versione Java è stato utilizzato il framework JUnit 4.8.2[20] che agevola la stesura e l’esecuzione dei test. Per quanto riguarda la versione per la piattaforma .NET è stato tradotto il motore di JUnit in .NET e poi sono stati eseguiti i test. Ciò permette di riutilizzare tutte le classi di test (TestCase) scritte in Java ma eseguendo di fatto i test sulla piattaforma .NET.

Di seguito sono riportati i risultati dei test per la versione 2.1.1 e 2.3.0alpha su piattaforma Java.

|  |  |  |
| --- | --- | --- |
|  | **2.1.1** | **2.3.0** |
| **Test standard** | 105 / 105 | 105 / 105 |
| **Test eccezioni** | - | 205 / 205 |

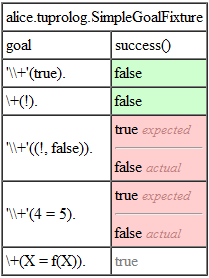
*Tabella 2 – Risultati test di unità piattaforma Java*

### Test di accettazione

I test di accettazione fanno parte di quella categoria di testing detta “Black-Box Testing” in cui il programma è considerato come una scatola nera ed è testato nella sua interezza, osservando la relazione “input-output”. Questi test non si basano quindi sulla struttura interna del sistema da testare ma piuttosto sulle specifiche di come deve funzionare.

Per quanto riguarda tuProlog, i test di accettazione testano tutti i predicati e funtori built-in e quelli messi a disposizione dalle librerie standard.

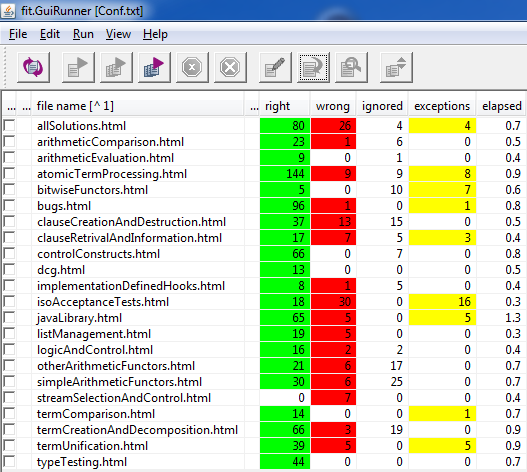
Il framework adottato per svolgere questo tipo di test è FIT (Framework for Integrated Test)[21]. L’utente fornisce degli esempi di corretto funzionamento del sistema formattati in tabelle e salvati in un documento HTML. Tali esempi vengono connessi dal programmatore al software da testare tramite le fixture. Durante l’esecuzione dei test viene controllato il documento, e creata una copia in cui: i test che hanno avuto successo sono colorati di verde, quelli che hanno lanciato eccezione di giallo e quelli che hanno fallito in rosso. Come per JUnit, per l’esecuzione dei test sulla piattaforma .NET è stato tradotto il motore di FIT.



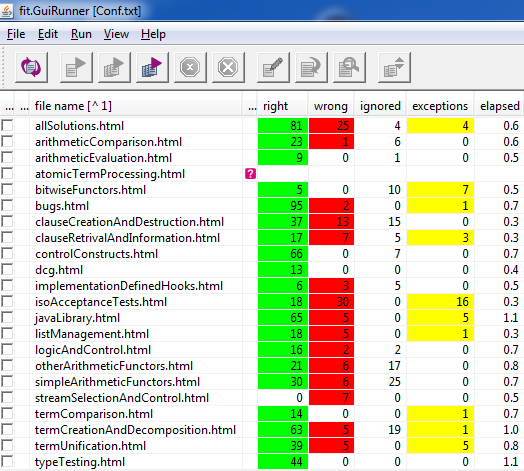
*Figura 3 – Esempio di tabella di output FIT*

Per automatizzare l’esecuzione dei test FIT è stato utilizzato lo strumento GUIRunner[21]. Tale strumento permette di eseguire in sequenza diversi test in automatico e ne mostra i risultati in un’unica tabella seguendo lo stesso schema di colori utilizzato nelle tabelle FIT (verde, giallo, rosso). Inoltre utilizzando GUIRunner è possibile impostare il comando per eseguire i test, pertanto può essere utilizzato anche per l’esecuzione dei test per la piattaforma .NET.

Di seguito sono riportati i risultati dei test per la versione 2.1.1 e 2.3.0alpha su piattaforma Java.



*Figura 4 – Risultati test di accettazione tuProlog 2.1.1 piattaforma Java*



*Figura 5 – Risultati test di accettazione tuProlog 2.3.0alpha piattaforma Java*

Come si nota dalle figure precedenti la versione Java dell’interprete non completa tutti i test con successo ma questo era noto dato che l’attuale implementazione non è priva di errori. L’obbiettivo della conversione verso la piattaforma .NET quindi, non è quello di risolvere tutti i problemi esistenti ma piuttosto fare in modo che la versione .NET dell’interprete presenti esattamente lo stesso comportamento della versione Java. Questo comporta che i test eseguiti sulla versione .NET debbano presentare un risultato del tutto identico a quelli della piattaforma Java, sia per quelli che hanno successo che per quelli che falliscono.

### Test delle prestazioni

Tali test sono stati realizzati per verificare la velocità di elaborazione dell’interprete. L’obbiettivo è quindi solo quello di confrontare le prestazioni, la correttezza del comportamento viene data per scontata.

Sono state realizzate due versioni degli stessi test: una per l’interprete sulla piattaforma Java e una per l’ambiente .NET. Lo schema di esecuzione rimane comunque invariato: viene letta da file una particolare teoria, viene stabilita l’obbiettivo da risolvere ed infine viene calcolato il tempo medio di esecuzione come differenza tra l'istante in cui inizia l'elaborazione e quello in cui finisce, diviso il numero di ripetizioni dell'operazione. Le ripetizioni per ogni test sono state 700.

Ogni test è costituito da una teoria (file *.pro*) e da una coppia di classi (Java, C#) che si occupano di lanciare e misurare i tempi di risposta del test sulle due diverse piattaforme.

I test utilizzati sono test standard che di solito vengono utilizzati per testare interpreti Prolog[23]:

* **Poly**: eleva simbolicamente un polinomio del tipo (1+x+y+z) alla decima potenza;
* **Crypt**: risolve un indovinello crypto-aritmetico;
* **Derive**: deriva simbolicamente quattro funzioni complesse di una variabile;
* **Nrev**: inverte una lista di 200 elementi;
* **Primes**: trova tutti i numeri primi fino a 100;
* **Qsort**: ordina una lista di 500 elementi utilizzando l’algoritmo QuickSort;
* **Queens**: trova come disporre 9 regine su una scacchiera 9x9 evitando che si “mangino” a vicenda;
* **Query**: trova i paesi con densità di popolazione approssimativamente uguale;
* **Tak**: benchmark artificiale, effettua moltissime invocazioni ricorsive e semplici operazioni aritmetiche;
* **I/O Write**: scrittura su file;
* **I/O Read**: lettura da file.

Di seguito sono riportati i risultati dei test per la versione 2.1.1 e 2.3.0 alpha su piattaforma Java.

|  |  |  |
| --- | --- | --- |
|  | **2.1.1**  **(millisecondi)** | **2.3.0alpha (millisecondi)** |
| **Poly** | 265 | 266 |
| **Crypt** | 18 | 18 |
| **Deriv** | 1 | 1 |
| **Nrev** | 360 | 361 |
| **Primes** | 20 | 21 |
| **Qsort** | 265 | 275 |
| **9 Queens** | 30 | 31 |
| **Query** | 20 | 20 |
| **Tak** | 1153 | 1150 |
| **I/O Write** | 30 | 27 |
| **I/O Read** | 103 | 76 |

*Tabella 3 – Risultati test delle prestazioni tuProlog 2.1.1 e 2.3.0 piattaforma Java*

## Approccio basato su IKVM.NET

### Risultati test di unità

|  |  |  |
| --- | --- | --- |
|  | **2.1.1** | **2.3.0** |
| **Test standard** | 105 / 105 | 105 / 105 |
| **Test eccezioni** | - | 205 / 205 |

*Tabella 4 – Risultati test di unità approccio IKVM.NET*

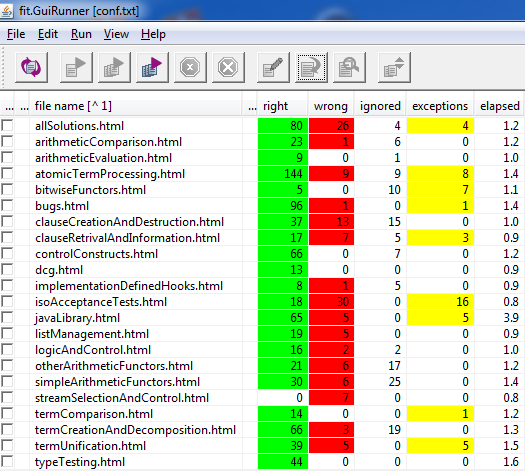
Come si può notare dalla Tabella 4 i test eseguiti sulla versione .NET tradotta con IKVM.NET forniscono gli stessi risultati di quelli eseguiti sulla versione standard.

L’unico problema incontrato durante l’esecuzione di tali test riguarda il fatto che, classi con visibilità di package in Java sono convertite in .NET con visibilità *internal*, limitandone pertanto la visibilità da parte di altri assembly. Ciò che accade è che i TestCase scritti in Java utilizzano alcune di queste classi, con visibilità di package, presenti nella distribuzione di tuProlog (*2p.jar*) ma quando si traduce in .NET questo non è più possibile: dall’assembly che contiene tutti i TestCase (*unit-tests.dll*) non sono visibili le classi *internal* all’interno dell’assembly di tuProlog (*2p.dll*). Per risolvere questo problema si è deciso di realizzare un unico assembly che contenesse sia le classi costituenti l’interprete sia i TestCase. Il comando utilizzato è il seguente:

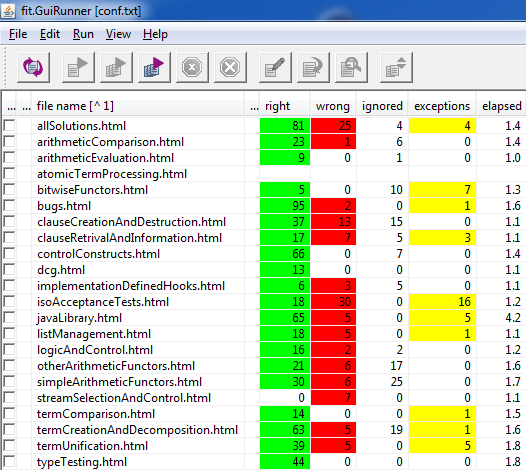
ikvmc -r:junit-4.exe -target:library -assembly:2pWithTests 2p.jar unit-tests.jar

I file *2p.jar* e *unit-tests.jar* sono gli archivi che contengono rispettivamente il motore tuProlog e i TestCase per i test di unità.

### Risultati test di accettazione



*Figura 6 – Risultati test di accettazione tuProlog 2.1.1 approccio IKVM.NET*



*Figura 7 – Risultati test di accettazione tuProlog 2.3.0 alpha approccio IKVM.NET*

Come si nota dalle figure precedenti, i risultati dei test di accettazione sono identici alla versione dell’interprete sulla piattaforma Java. Questo dimostra che l’approccio di traduzione automatica basato su IKVM.NET non introduce alterazioni nel comportamento dell’interprete. Ciò conferma l’efficacia di tale strumento per la conversione automatica di applicazione Java su piattaforma .NET.

Per eseguire questi test sono sorti alcuni problemi relativi il class loading: nei file HTML è specificato il nome di una classe fixture nel formato Java (esempio alice.tuprolog.SimpleGoalFixture) ma l’assembly preposto al caricamento ed esecuzione dei test (*fit.exe*) non avendo un riferimento all’assembly contenente le fixtures (*fixtures.dll*) non è in grado di caricare le relative classi. Per risolvere il problema si potrebbe operare in tre modi (sezione 4.7.1):

1. generare un unico assembly che contenga sia il motore FIT che le fixtures relative a test di tuProlog;
2. tradurre il motore FIT specificando come class loader il ClassPathAssemblyClassLoader e inserire nella stessa directory in cui vengono eseguiti i test, le classi fixture compilate in Java (file *.class*);
3. tradurre il motore FIT specificando come class loader l’AppDomainAssemblyClassLoader e realizzare un’applicazione (*NETFileRunner.exe*) che si occupi di caricare, nell’AppDomain, le classi fixtures, contenute in un’assembly passato come parametro e poi esegua i test.

La soluzione 1 non permette di riutilizzare il motore FIT convertito in .NET per eseguire altri test dato che è in grado di caricare solo le fixture contenute all’interno del suo assembly (*fit.exe*). La soluzione 2 potrebbe risultare caotica dato che nella stessa directory occorre inserire la versione .NET del motore FIT (*fit.exe*) e tutte le classi fixtures (*.class*), quindi ci sarebbero alcuni file .NET e altri Java. Per questi motivi sono state scartate le prime due soluzioni. La soluzione 3 invece è più semplice perché è sufficiente avere nella stessa directory solo tre assembly: *fit.exe*, *2p.exe*, *fixtures.dll*, rispettivamente contenenti il motore FIT, il motore tuProlog e le classi fixtures. Inoltre l’applicazione *NETFileRunner.exe* può essere riutilizzata per eseguire qualunque test specificando un diverso assembly che contiene le classi fixtures.

Per dettagli sull’utilizzo di *NETFileRunner.exe* fare riferimento all’appendice E.

### Risultati test delle prestazioni

Di seguito sono riportati i risultati dei test sulle prestazioni per l’interprete tradotto in .NET con IKVM.NET.

|  |  |  |
| --- | --- | --- |
|  | **2.1.1**  **(millisecondi)** | **2.3.0alpha (millisecondi)** |
| **Poly** | 325 | 319 |
| **Crypt** | 20 | 21 |
| **Deriv** | 1 | 1 |
| **Nrev** | 489 | 488 |
| **Primes** | 25 | 26 |
| **Qsort** | 413 | 349 |
| **9 Queens** | 36 | 37 |
| **Query** | 22 | 23 |
| **Tak** | 1316 | 1323 |
| **I/O Write** | 24 | 20 |
| **I/O Read** | 92 | 74 |

*Tabella 5 – Risultati test delle prestazioni tuProlog 2.1.1 e 2.3.0 approccio IKVM.NET*

Il tempo aggiuntivo, rispetto alla versione Java, per eseguire tutti i test è mediamente pari al 14,9% per la versione 2.1.1 e 13,0% per la versione 2.3.0alpha.

## Approccio basato su Ja.NET

Poiché Ja.NET non implementa il package org.omg.CORBA, la versione 2.1.1 di tuProlog è stata compilata in .NET escludendone la parte che implementa il servizio CORBA. Non è stata possibile compilare la versione 2.3.0alpha dato che utilizza il package non implementato com.sun.jdi. Pertanto tutti i test sono stati condotti solo sulla versione 2.1.1.

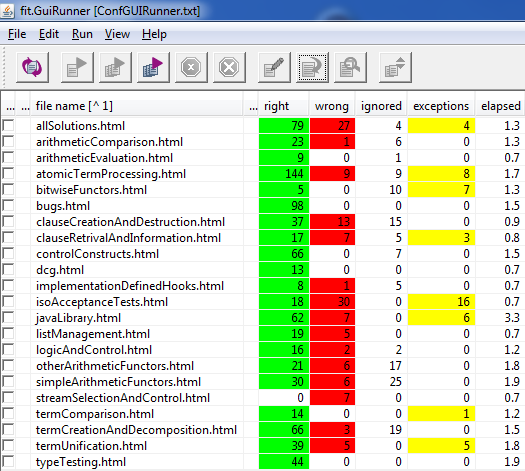
### Risultati test di unità

|  |  |
| --- | --- |
|  | **2.1.1** |
| **Test standard** | 105 / 105 |

*Tabella 4 – Risultati test di unità approccio Ja.NET*

Utilizzando Ja.NET non si incorre nello stesso problema della visibilità delle classi incontrato con IKVM.NET perché è sufficiente compilare i sorgenti di tuProlog senza aggiungere l’opzione secureScope (sezione 2.4.2), in questo caso, infatti, le classi con visibilità di package vengono compilate in classi con visibilità pubblica, quindi utilizzabili anche da assembly esterni.

### Risultati test di accettazione



*Figura 8 – Risultati test di accettazione approccio Ja.NET*

Dalla Figura 8 è possibile notare come alcuni test (*allSolutions.html*) falliscano in numero maggiore rispetto a quelli eseguiti sulla versione Java. Ciò dimostra l’incapacità di Ja.NET di supportare un processo automatico di conversione di applicazioni Java su piattaforma .NET.

### Risultati test delle prestazioni

|  |  |
| --- | --- |
|  | **2.1.1**  **(millisecondi)** |
| **Poly** | 463 |
| **Crypt** | 41 |
| **Deriv** | 1 |
| **Nrev** | 520 |
| **Primes** | 38 |
| **Qsort** | 409 |
| **9 Queens** | 54 |
| **Query** | 41 |
| **Tak** | 2421 |
| **I/O Write** | 14 |
| **I/O Read** | 50 |

*Tabella 5 – Risultati test delle prestazioni tuProlog 2.1.1 e 2.3.0 approccio IKVM.NET*

Il tempo aggiuntivo, rispetto alla versione Java, per eseguire tutti i test è mediamente pari al 52,9%.

## Conclusione

Come si evince dai risultati dei test la soluzione di porting offerta da IKVM.NET risulta migliore sotto ogni punto di vista: per quanto riguarda il comportamento dell’interprete, infatti, i test forniscono gli stessi risultati della versione Java. Per quanto riguarda le prestazioni si ha un prezzo da pagare (+15% circa) che è piccolo se confrontato con il fatto che è possibile ottenere la versione per la piattaforma .NET senza scrivere alcun codice di adattamento. Inoltre non va sottovalutato che con IKVM.NET l’interprete per l’ambiente .NET è completo di interfaccia grafica.

I test effettuati sulla versione compilata con Ja.NET forniscono ulteriori prove sulla inadeguatezza di tale approccio per effettuare il porting di applicazioni di una certa complessità. Infatti la versione .NET ottenuta con tale approccio sarebbe stata privata di alcune funzionalità importanti come l’interfaccia grafica e l’utilizzo come servizio remoto (CORBA). Inoltre Ja.NET introduce un discreto overhead nell’esecuzione dell’applicazione che porta ad avere delle prestazioni troppo distanti dal riferimento imposto dalla versione Java.

# Supporto all’interoperabilità per linguaggi .NET

Come discusso nel capitolo 4, tuProlog permette una completa integrazione Java/Prolog e Prolog/Java, grazie alla libreria JavaLibrary. Con l’introduzione della nuova versione per la piattaforma .NET si è resa disponibile l’integrazione tra qualsiasi linguaggio .NET e Prolog. Infatti, come mostrato nell’appendice F, è possibile istanziare ed utilizzare un motore tuProlog all’interno di un progetto .NET. Quello che non si è ottenuto automaticamente in seguito alla conversione è la possibilità di utilizzare componenti .NET da Prolog. La libreria JavaLibrary, infatti, è stata progettata tenendo in considerazione le peculiarità del linguaggio Java (costrutti disponibili e convenzione di scrittura), dato che l’obbiettivo era l’interoperabilità con quest’ultimo. Quindi anche se la JavaLibrary è stata tradotta in .NET di fatto, per come è stata implementata, non è in grado di supportare tutte le caratteristiche dei diversi linguaggi .NET (ad esempio le proprietà). I linguaggi disponibili per la piattaforma .NET sono infatti innumerevoli e ciascuno presenta delle proprie peculiarità. Inoltre durante la fase di traduzione in CIL, vengono introdotti adattamenti necessari per poter esprimere le diverse caratteristiche nel linguaggio intermedio.

Pertanto, è stata progettata e realizzata una libreria tuProlog (OOLibrary) che consenta l’utilizzo di tutte le caratteristiche offerte dai linguaggi .NET da Prolog, nascondendo all’utente le conversioni introdotte dal compilatore.

## Analisi del problema

In seguito alla conversione in .NET della libreria JavaLibrary e all’estensione della semantica del metodo Class.forName(String) (sezione 1.5.3)introdotta da IKVM.NET è possibile utilizzare oggetti .NET con gli stessi predicati utilizzati per le entità Java, ma con alcuni problemi.

I problemi riscontrati sono principalmente tre:

1. alcuni costrutti linguistici, come ad esempio le proprietà, non sono disponibili in Java, quindi la JavaLibrary non fornisce un supporto per l’utilizzo di tali strumenti;
2. la convenzione sui nomi degli elementi sintattici utilizzata dal linguaggio Java è diversa dalle convenzioni adottate dai principali linguaggi .NET. Ad esempio Java prevede che i nomi dei metodi inizino con la lettera minuscola e seguano il CamelCase mentre C# prevede che inizino con la lettera maiuscola;
3. i compilatori per la piattaforma .NET compilano i sorgenti riorganizzando il codice e in alcuni casi modificando i nomi degli elementi sintattici. Quindi ciò che si verifica è che lo sviluppatore scrive un codice ma di fatto quello che viene memorizzato nell’assembly in CIL è diverso.

Di seguito sono riportati alcuni esempi che sottolineano le modifiche da effettuare per utilizzare un oggetto .NET attraverso la JavaLibrary (il codice della classe *Student* è riportato in Figura 9):

* **creazione di un oggetto**

java\_object('CStudent.Student, CStudent',[311471,'ale','monta'], Obj),

E’ possibile istanziare un oggetto .NET specificando l’AssemblyQualifiedNamedella classe, eventualmente anche con la specifica di versione, cultura e chiave pubblica.

* **Invocazione di un metodo di istanza**

Obj <- 'PrintStudent' returns Value,

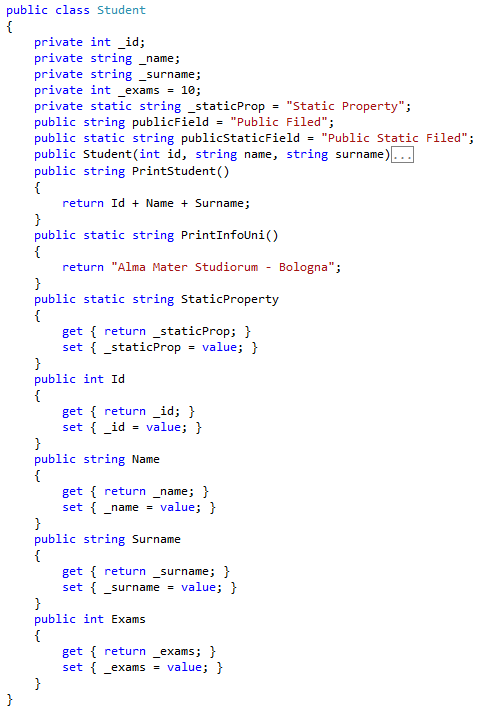
Il nome del metodo deve essere specificato tra apici singoli (atomo) per indicare il nome corretto secondo la convenzione dello specifico linguaggio in cui è stato scritto.

* **utilizzo di una proprietà**

Obj <- 'get\_Name' returns Value,

Obj <- 'set\_Name'('Paolo'),

La JavaLibrary non è in grado di utilizzare le proprietà .NET nativamente; tuttavia dato che queste ultime vengono compilate (dai compilatori .NET) in metodi con prefisso “get\_” oppure “set\_” è possibile utilizzarle attraverso la JavaLibrary indicando il nome del corrispondente metodo. In questo caso si ha una perdita di espressività, in quanto si utilizza una proprietà come se fosse un metodo e questa non è una pratica comune tra gli sviluppatori .NET che preferiscono la notazione: Oggetto.Proprietà.



*Figura 9 – Codice classe Student utilizzata negli esempi dei predicati (C#)*

## OOLibrary

Ciò che si evince dagli esempi della sezione precedente è la possibilità di utilizzare un qualsiasi oggetto .NET attraverso la JavaLibrary tradotta con IKVM.NET senza modificare la libreria. Il problema consiste nel fatto che è necessario conoscere tutte le alterazioni introdotte dal compilatore (vedi proprietà) per poter utilizzare tutti gli elementi specificando il nome corretto.

Al fine di evitare all’utente di dover conoscere le modifiche introdotte dal compilatore .NET è necessario introdurre uno strato software che si occupi di nascondere all’utente le modifiche del compilatore, consentendogli di utilizzare la convenzione standard del linguaggio di programmazione sorgente.

Queste motivazioni hanno portato alla realizzazione della OOLibrary: libreria tuProlog che consente di interagire con componenti software scritti sia in linguaggio Java che nei principali linguaggi .NET. In prima applicazione, i linguaggi .NET supportati sono C#, F# e VB.NET.

La OOLibrary è stata inoltre realizzata tenendo conto delle possibili estensioni riguardanti i linguaggi supportati: in particolare si è cercato di semplificare l’introduzione del supporto per un nuovo linguaggio .NET.

### Convenzioni

Per nascondere le caratteristiche di ogni singolo linguaggio si è deciso di utilizzare il concetto di *Convenzione*, già introdotto da Marco Albertin nel suo lavoro di tesi[19].

Le Convenzioni sono utilizzate dalla OOLibrary come supporto per la conversione dei nomi degli elementi sintattici. Ogni Convenzione, infatti, espone dei metodi che hanno lo scopo di fornire il nome dell’entità richiesta (esempio una classe, un metodo, una proprietà, un campo pubblico) opportunamente modificato in base alle alterazioni introdotte dal compilatore. In tal modo l’utente può richiamare una classe o un membro specificando il nome originale, cioè quello utilizzato durante la stesura del codice, senza conoscere le modifiche apportate dal compilatore.

Utilizzando le Convenzioni, inoltre, la OOLibrary non è legata staticamente a particolari linguaggi, quindi è possibile realizzare il supporto per nuovi linguaggi senza modificare la libreria: infatti, è sufficiente creare una nuova Convenzione che ridefinisca le operazioni necessarie per eseguire le opportune conversioni. In tal modo il sistema è estendibile senza problemi.

### Interfaccia

Prima della realizzazione della libreria vera e propria sono stati definiti i predicati e gli operatori che la OOLibrary deve esporre al motore tuProlog e il loro comportamento.

I predicati previsti sono i seguenti:

* load\_convention/3: questo predicato si occupa di caricare una particolare convenzione e di associarla ad un termine Prolog. La sintassi da utilizzare è la seguente:

load\_convention(*assemblyName*, *className* , *conventionID*)

dove *assemblyName* rappresenta il nome del file assembly che contiene la convenzione, *className* indica il nome della classe della convenzione e *conventionID* rappresenta la variabile alla quale verrà associata la convenzione. Quest’ultimo può essere passato come parametro del predicato new\_object/4descritto in seguito. Ogni convenzione caricata mediante questo predicato ha una validità pari a quella della query correntemente eseguita. Per il caricamento delle convenzioni è stata prevista anche la direttiva load\_convention/3che presenta lo stesso comportamento del predicato omonimo, con l’unica differenza che le convenzioni caricate in questo modo hanno una validità pari a quella del motore tuProlog corrente.

* unload\_convention/1: questo predicato è utilizzato per scaricare una particolare convenzione in accordo con la seguente sintassi:

unload\_convention(*conventionID*)

* new\_object/4: tale predicato viene utilizzato per creare un nuovo oggetto della classe specificata utilizzando una particolare convenzione. La sintassi da utilizzare è la seguente:

new\_object(*conventionName*, *className*, *argList*, *objRef*)

Il parametro conventionName contiene il nome della convenzione da utilizzare per istanziare l’oggetto (può essere specificata passando il termine conventionID restituito dal predicato per il caricamento della convenzione). Se la convenzione è già stata caricata, viene innanzitutto utilizzata per istanziare l’oggetto e successivamente il nome della classe istanziata viene associato alla convenzione, per fare in modo che successive operazioni sull’oggetto siano mediate da quella particolare convenzione. Se invece la convenzione non è caricata (oppure non è specificato il nome), si suppone che l’utente conosca esattamente i nomi delle entità da utilizzare e quindi non viene effettuata nessuna conversione sui nomi. Il parametro className indica il nome della classe che deve essere istanziata. Nel caso in cui si voglia istanziare un oggetto .NET occorre specificare il nome nel formato AssemblyQualifiedName: NameSpace.NomeClasse, NomeAssembly, Versione, Cultura, ChiavePubblica. L’unica informazione indispensabile oltre al nome della classe è NomeAssembly. argList rappresenta la lista dei parametri da passare al costruttore e objRef rappresenta il termine Prolog al quale viene associato l’oggetto istanziato. Con questo predicato è possibile istanziare un array della classe specificata, indicando dopo il nome della classe i caratteri “[]” che usualmente indicano un array, oppure i caratteri consentiti dal linguaggio nel caso in cui si utilizzi la convenzione specifica.

* destroy\_object/1: con questo predicato è possibile rimuovere l’associazione tra il termine Prolog specificato come parametro (objRef) e l’oggetto precedentemente legato al termine. La sintassi da utilizzare è la seguente:

destroy\_object(objRef)

* method\_call/3: con questo predicato è possibile invocare un metodo su un oggetto utilizzando la seguente sintassi:

method\_call(objRef, methodInfo, objResultRef)

dove objRefrappresenta il termine Prolog a cui è associato l’oggetto sul quale invocare il metodo e methodInfoindica un termine composto rappresentate sia il nome del metodo che gli eventuali parametri. objResultRefrappresenta il termine a cui viene associato l’eventuale valore di ritorno del metodo. Tale predicato prima dell’invocazione tenta di ricavare la convenzione per l’oggetto specificato. Se la convenzione viene trovata si applicano le relative conversioni prima di invocare il metodo. Se invece la convenzione non viene trovata si suppone che l’utente conosca esattamente il nome del metodo e quindi non viene fatta nessuna conversione.

* array\_set/3: con questo predicato è possibile inserire un oggetto all’interno di un array nella posizione specificata. La sitassi da utilizzare è:

array\_set(objArrayRef, index, objRef)

dove objArrayRefindica un termine a cui è associato un array, indexè un intero che indica la posizione in cui inserire l’oggetto objRef.

* array\_get/3: con questo predicato è possibile ottenere l’oggetto presente in una particolare posizione di un array con la sintassi:

array\_get(objArrayRef, index, objResultRef)

I parametri objArrayRefe indexindicano rispettivamente l’array da cui prelevare l’oggetto e la posizione. objResultRefindica il termine a cui verrà associato l’eventuale oggetto presente nella posizione specificata.

* array\_length/3: ritorna valore vero se l’array passato come parametro (objArrayRef) ha la dimensione indicata dal secondo parametro (arrayLength).

array\_length(objArrayRef, arrayLength)

La OOLibrary definisce anche due operatori:

* ‘<-‘: questo operatore permette di invocare un metodo su un oggetto secondo la seguente sintassi:

objRef <- methodName( arguments )

objRef <- methodName( argList ) returns value

dove objRefindica un termine Prolog al quale è associato un oggetto, methodNamerappresenta il nome del metodo da invocare e argumentssono gli eventuali parametri del metodo. Utilizzando la parola chiave returnsè possibile associare al termine valuel’eventuale valore di ritorno del metodo. Questo operatore viene utilizzato risolvendo il predicato method\_call/3descritto precedentemente, quindi viene applicata la convenzione se esistente.

* ‘.’: con questo operatore (e con la coppia di pseudo metodi get/set) è possibile accedere a campi pubblici o proprietà dell’oggetto specificato. La sintassi da utilizzare è la seguente:

objRef.field <- get(value)

objRef.field <- set(newValue)

objRef.property <- get(value)

objRef.property <- set(newValue)

dove objRefindica un termine Prolog al quale è associato un oggetto, fielde propertyindicano il nome del campo pubblico o della proprietà. valueindica il termine al quale viene associato il valore restituito dal campo o dalla proprietà e newValue il nuovo valore per il campo o la proprietà (può essere un dato primitivo o un riferimento ad un oggetto).Anche questo operatore viene utilizzato risolvendo il predicato method\_call/3.

Per implementare la libreria: le soluzioni prese in considerazione hanno riguardato sia un approccio Java-based, cioè soluzione basata sulla piattaforma Java, che .NET-based, cioè soluzione basata sull’ambiente .NET.

Peraltro, la OOLibrary ha senso di esistere solo nella versione .NET dell’interprete tuProlog, poiché essa è l’unica a permettere l’interazione tra Prolog e i linguaggi .NET e l’utente ha bisogno delle convenzioni solo quando utilizza oggetti scritti in uno dei linguaggi .NET: nella versione Java, infatti, non occorrono convenzioni, dato che il linguaggio supportato è unico.

### Possibili soluzioni in Java

Le soluzioni prese in considerazione sulla piattaforma Java sono due:

1. **Modifica della JavaLibrary**: questa soluzione prevede di modificare il codice esistente della JavaLibrary per introdurre il concetto di Convenzione. Innanzi tutto, devono essere aggiunti i predicati necessari per il caricamento e scaricamento delle convenzioni. In secondo luogo l’implementazione esistente dei predicati per l’invocazione di metodi e per l’utilizzo di campi pubblici deve essere modificata per supportare le convenzioni. Questo approccio rende necessaria la modifica di una grande quantità di codice già funzionante, con il rischio di introdurre nuovi errori. Inoltre non è da sottovalutare il fattore tempo: integrare nuove funzionalità in un’applicazione già completa, infatti, non è un’operazione veloce, perché molto probabilmente il sistema non è stato progettato per supportare un’estensione di questo tipo. Quindi c’è un’alta probabilità di dover adattare l’intero sistema in seguito alla modifica.
2. **Subclassing della JavaLibrary**: tale soluzione prevede la realizzazione di una nuova libreria che derivi direttamente dalla JavaLibrary. In questo modo tutto il codice della JavaLibrary non deve essere modificato, superando, di fatto, le principali limitazioni della soluzione precedente. L’implementazione dei nuovi predicati (sezione 5.2.2) si limiterebbe a convertire i nomi delle entità (metodi, campi, proprietà) in base all’eventuale convenzione e ad invocare il corrispondete predicato della classe base, cioè della JavaLibrary. In tal modo eventuali modifiche apportate per migliorare la JavaLibrary verrebbero adottate automaticamente anche dalla OOLibrary.

Poiché in questa ipotesi la OOLibrary deriverebbe dalla JavaLibrary, erediterebbe tutti i metodi di quest’ultima, quindi gli utenti che caricassero in un motore tuProlog la OOLibrary potrebbero utilizzare sia i nuovi predicati che si appoggiano sulle convenzioni sia i predicati classici della JavaLibrary che invece non ne fanno uso.

Nel caso di soluzioni di questo tipo la nuova libreria sarebbe inserita nella versione per la piattaforma Java di tuProlog e poi convertita in .NET con IKVM.NET.

Dunque, entrambe queste soluzioni basate su Java appaiono insoddisfacenti, in quanto: viene introdotto il concetto di convenzione nella versione Java dell’interprete, anche se non necessario. In questa versione, infatti, non vi è la necessità di convenzioni dato che il linguaggio supportato è unico. Quindi si complicherebbe la versione Java soltanto con la prospettiva di ottenere una nuova funzionalità nella versione .NET. Un altro aspetto da non sottovalutare è quello riguardante le prestazioni: infatti, come discusso nella sezione 4.7.3, il processo di traduzione in CIL operato da IKVM.NET introduce un calo di prestazioni di circa il 14%. Quindi dato che le convenzioni sono necessarie solo nella versione .NET dell’interprete, risulta conveniente implementare la OOLibrary direttamente in .NET per ottenere le massime prestazioni possibili.

### Possibili soluzioni in .NET

Le soluzioni analizzate per la realizzazione della OOLibrary direttamente sulla piattaforma .NET sono due:

1. **Proxy**: secondo tale soluzione la OOLibrary dovrebbe svolgere il ruolo di proxy verso la JavaLibrary, definendo i nuovi predicati come da specifiche (sezione 5.2.2). La nuova libreria dovrebbe inoltre mantenere un riferimento alla JavaLibrary, in modo che ogni operazione richiesta alla OOLibrary sia preventivamente elaborata per modificare i nomi in base all’eventuale convenzione e successivamente delegata alla JavaLibrary incapsulata. Con questa soluzione si risolverebbero i problemi di prestazioni citati nella sezione precedente dato che la OOLibrary sarebbe implementata direttamente in .NET.

Analizzando nel dettaglio questa soluzione è tuttavia emerso un problema che ne impedisce di fatto l’applicazione concreta: ogni libreria, infatti (quindi anche la JavaLibrary), necessita di un riferimento ad un motore tuProlog, per funzionare correttamente. Poiché tale riferimento viene aggiunto al momento del caricamento della libreria mediante un metodo con visibilità di package, che viene tradotto da IKVM.NET con visibilità internal(quindi inaccessibile) (sezione 1.5.2) la JavaLibrary incapsulata all’interno della OOLibrary non risulta in grado di funzionare correttamente perché è impossibile associarla ad un motore tuProlog.

1. **Subclassing della JavaLibrary**: questa soluzione è del tutto analoga all’omonima soluzione analizzata nella sezione precedente. La differenza principale risiede nel fatto che in questo caso la nuova libreria sarebbe scritta direttamente in .NET quindi non si avrebbero cali di prestazioni introdotti dalla successiva conversione, come accade nella soluzione Java.

### Confronto

Per scegliere la soluzione da adottare per implementare la OOLibrary sono stati tenuti in considerazione i seguenti aspetti:

* **semplicità di implementazione**: deve essere possibile realizzare la OOLibrary senza troppa difficoltà, concentrandosi soprattutto sulla gestione delle convenzioni;
* **tempo necessario per l’implementazione**: per implementare la nuova libreria deve essere necessario un tempo relativamente breve;
* **manutenibilità nel tempo**: deve essere semplice modificare ed estendere la nuova libreria;
* **prestazioni**: si devono ricercare le migliori prestazioni possibili.

Tenendo conto di questi aspetti la soluzione peggiore è quella che consiste nel modificare direttamente il codice della JavaLibrary in quanto sarebbe un’operazione molto lunga e difficile, con alta probabilità di introdurre nuovi errori. Inoltre il codice per gestire le convenzioni risulterebbe “sparpagliato” all’interno delle attuali implementazioni dei predicati, rendendo difficile apportare modifiche al sistema di gestione delle convenzioni. Oltretutto la traduzione in .NET comporterebbe un aumento del tempo di esecuzione, peggiorando le prestazioni.

Con la soluzione di subclassing sulla piattaforma Java, invece, la parte di gestione delle convenzioni sarebbe implementata nella OOLibrary. Quindi si verrebbe a creare una netta distinzione tra la gestione vera e propria dei predicati (contenuta nella JavaLibrary) e la gestione dei nomi e delle convenzioni (contenuta nella OOLibrary) semplificando notevolmente l’eventuale modifica di quest’ultima. L’unico difetto sostanziale di questo approccio riguarda le prestazioni, che però, come detto, si risolve implementando la OOLibrary direttamente in .NET con la soluzione del subclassing.

Tale scelta porta ai benefici già discussi per la soluzione di subclassing su Java con in più il miglioramento dovuto al mancato calo di prestazioni.

Di seguito è riportata una tabella che sintetizza i punti di forza e le debolezze delle soluzioni analizzate. Ogni asterisco indica un punto favorevole per la soluzione interessata.

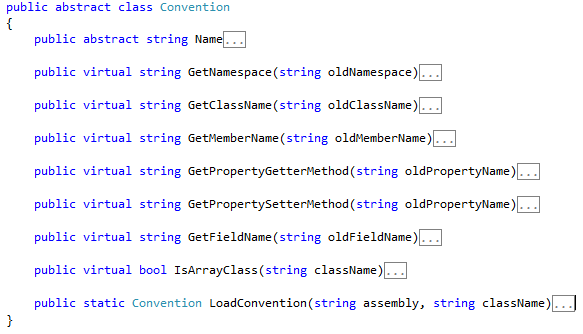
|  |  |  |  |
| --- | --- | --- | --- |
|  | **Modifica JavaLibrary** | **Subclassing JavaLibrary (Java)** | **Subclassing JavaLibrary (.NET)** |
| Semplicità | \* | \*\*\*\* | \*\*\*\* |
| Tempo | \* | \*\*\* | \*\*\* |
| Manutenibilità | \* | \*\*\*\* | \*\*\*\* |
| Prestazioni | \*\* | \*\* | \*\*\*\*\* |

*Tabella 6 – Confronto soluzioni implementative OOLibrary*

### Implementazione delle convenzioni

Data la scelta di realizzare la OOLibrary direttamente in .NET, è stato scelto il linguaggio C#.

Come si nota dalla Figura 10 la classe Conventionè astratta. Questa scelta è dettata dall’esigenza di impostare un comportamento predefinito per tutte le operazioni di conversione. In particolare il comportamento prefissato non modifica il nome passato come parametro, quindi è come se non venisse applicata nessuna conversione sul nome. Rispetto alla soluzione di Albertin[19] il concetto di Convenzione è stato semplificato: nella soluzione precedente, infatti, la convenzione espone funzionalità per il recupero, l’istanziazione e l’utilizzo delle classi contenute in un assembly; tali funzioni sono utilizzate dalla CLILibrary (libreria per l’interoperabilità tra linguaggi .NET). In questa soluzione invece la convenzione si occupa solo della modifica dei nomi delle entità utilizzate; il recupero e l’utilizzo viene delegato alla JavaLibrary.



*Figura 10 – Implementazione classe Convention*

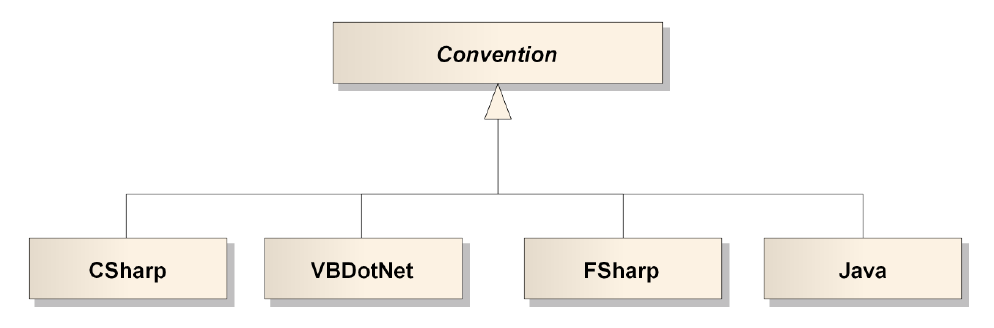
Nel dettaglio i metodi di conversione sono i seguenti:

* GetNamespace: converte il nome del namespace;
* GetClassName: converte il nome della classe;
* GetMethodName: converte i nomi dei metodi;
* GetPropertyGetterMethod: converte il nome della proprietà nel corrispondente metodo get;
* GetPropertySettermethod: converte il nome della proprietà nel corrispondente metodo set;
* GetFieldName: converte il nome del campo.

Il metodo IsArrayClassè necessario per capire se il nome della classe passato come parametro identifica un array. Il comportamento predefinito prevede di verificare che il nome della classe termini con “[]” che è il suffisso tipicamente usato per un array (ma può essere ridefinito da una particolare convenzione).

La classe Conventioncontiene anche un metodo statico (LoadConvention)utilizzato per caricare una convenzione e una proprietà (Name) utilizzata per identificare una particolare Convenzione. Tale proprietà (astratta) deve essere ridefinita dalle sottoclassi e dovrebbe contenere il nome del linguaggio per il quale vale la convenzione corrente (esempio “csharp”, “fsharp”).

Tutte le convenzioni per i diversi linguaggi devono derivare dalla classe astratta Conventionper fare in modo che la OOLibrary sia in grado di caricarle. Quindi in generale si verrà a creare una gerarchia di classi in cui la radice è occupata da Convention mentre le classi derivate rappresentano le vere convenzioni per i diversi linguaggi supportati.



*Figura 11 – Gerarchia delle convenzioni*

Sono state realizzare le convenzioni per i seguenti linguaggi:

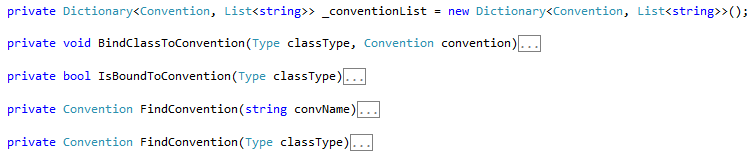
* **C#**: in tale linguaggio i nomi delle entità, eccetto quelli dei campi, devono iniziare con la lettera maiuscola, quindi tutti i nomi delle classi, dei metodi e dei namespace devono essere modificati di conseguenza. I nomi dei campi invece devono iniziare con la lettera minuscola. Per quanto riguarda le proprietà, dato che il compilatore le traduce in metodi con prefisso “get\_” e “set\_” i relativi metodi (GetPropertyGetterMethode GetPropertySetterMethod) devono restituire delle stringhe nel formato “get\_NomeProp” e “set\_NomeProp”, dove NomeProp deve iniziare con la lettera maiuscola.
* **VB.NET**: per quanto riguarda i nomi presenta un comportamento identico alla convenzione C#. Inoltre dato che in Visual Basic gli array si indicano terminando il nome della classe con i caratteri”()” e non con “[]”, il metodo IsArrayClass è stato ridefinito per supportare questa caratteristica.
* **F#**: vengono applicate le stesse conversioni utilizzate nella convenzione C#;
* **Java**: questa convenzione modifica i nomi dei metodi e dei campi facendoli iniziare con la lettera minuscola, i nomi delle classi con la lettera maiuscola, mentre i package vengono convertiti completamente in minuscolo.

### Implementazione della libreria

Come discusso nella sezione 5.2.5 si è deciso di implementare la OOLibrary direttamente in .NET e con la tecnica del subclassing.

La OOLibrary deve presentare un comportamento comune a prescindere dai linguaggi utilizzati, quindi non utilizza mai una particolare convenzione ma soltanto la classe astratta Convention. In questo modo è possibile implementare la libreria senza curarsi di quale convenzione sarà utilizzata durante l’esecuzione.

Ogni volta che viene caricata una convenzione quest’ultima viene inserita, come chiave, all’interno di un dizionario (Figura 12). Il dizionario \_conventionListmantiene i riferimenti a tutte le convenzioni caricate, sotto forma di chiavi del dizionario stesso, e per ciascuna mantiene, sotto forma di valore, una lista di stringhe per memorizzare i nomi di tutte le classi che devono essere mediate da quella particolare convenzione. Quindi ogni volta che viene istanziato un nuovo oggetto con il predicato new\_object/4e viene specificata una convenzione, sarà inserita una nuova entry all’interno del dizionario. Per la gestione delle convenzioni, sono stati implementati alcuni metodi di supporto (Figura 12). BindClassToConventione IsBoundToConvention si occupano rispettivamente di associare il nome di una classe ad una determinata convenzione, per fare in modo che qualunque operazione compiuta sulla classe sia mediata da quella particolare convenzione e controllare se una classe è già stata associata ad una convenzione. Il metodo FindConvention si occupa invece di trovare la convenzione specificando il nome di quest’ultima oppure una classe (classType); in quest’ultimo caso verrà restituita l’eventuale convenzione associata alla classe passata come parametro



*Figura 12 – Supporto per le convenzioni*

Dato che il motore tuProlog utilizza la reflection per individuare i predicati e le direttive, occorre seguire una certa signature quando si implementano.

La signature da seguire è la seguente:

* **Predicati**

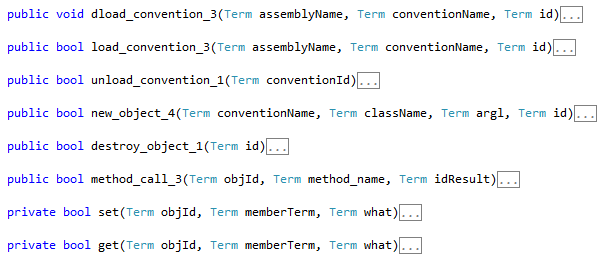
public bool <pred name>\_<N>(T1 arg1, T2 arg2, ...,Tn argN)

* **Direttive**

public void <dir name>\_<N>(T1 arg1, T2 arg2, ..., Tn argN)

dove T1, T2, …, Tnrappresentano gli eventuali parametri della direttiva o predicato.

In accordo con la sintassi richiesta sono stati implementati i metodi che sono invocati quando è richiesta la risoluzione dei corrispondenti predicati o direttive (Figura 13).



*Figura 13 – Predicati OOLibrary*

Dato che è stata scelta la soluzione del subclassing i metodi si limitano a modificare i nomi dell’entità passate come parametri (esempio metodi, campi pubblici, ecc.) e ad invocare i corrispondenti metodi della JavaLibrary. Quindi ad esempio nel caso in cui venga invocato un metodo su un oggetto scritto in linguaggio C#, la OOLibrary eseguirà i seguenti passi:

1. ricava la convenzione associata all’oggetto (se esiste);
2. modifica il nome del metodo mediante la convenzione;
3. invoca il metodo java\_call\_3(…)della classe base.

In questo modo si ottiene la netta divisione di responsabilità, accennata nella sezione 5.2.5: la OOLibrary si occupa di modificare i nomi, la JavaLibrary si occupa di interagire effettivamente con gli oggetti (invocazione metodi, assegnamento valori, ecc.)

Il metodo dload\_convention\_3(…)rappresenta la direttiva per il caricamento delle convenzioni.

I metodi set(…)e get(…) sono metodi di supporto che vengono invocati da method\_call\_3(…)nel caso un cui sia necessario assegnare o recuperare il valore di un campo pubblico o di una proprietà. Tali metodi sono privati e quindi non possono essere utilizzati dall’esterno. Le proprietà hanno un meccanismo di assegnazione e di recupero del valore identico a quello dei campi pubblici, quindi i metodi set(…) e get(…) devono gestire entrambi i casi. Innanzi tutto controllano la presenza del campo pubblico e successivamente della proprietà; solo una delle due operazioni di ricerca può avere esito positivo dato che non è possibile definire, all’interno della stessa classe, una proprietà e un campo con lo stesso nome.

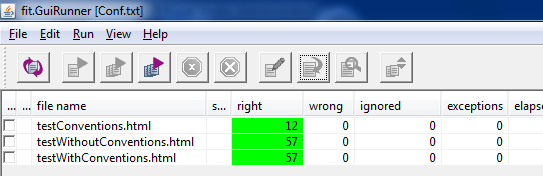
### Collaudo

Il collaudo della OOLibrary è stato realizzato in due fasi:

1. **test delle convenzioni**: inizialmente sono state testate le singole convenzioni per verificare che i nomi fossero convertiti in accordo con la specifica convenzione del linguaggio. Successivamente sono stati testati i predicati per il caricamento e scaricamento delle convenzioni.
2. **test di utilizzo**: in questa fase è stato controllato il funzionamento della libreria nella sua interezza. L’obbiettivo di questa fase è stato quello di verificare che tutti i predicati presentino il comportamento indicato nelle specifiche riportate nella sezione 5.2.2. Tale verifica si è svolta secondo due modalità: nella prima sono stati utilizzati alcuni oggetti (.NET e Java) da Prolog senza utilizzare le specifiche convenzioni; nella seconda invece sono state effettuate le stesse prove con l’utilizzo delle convenzioni.

I test sulle singole convenzioni sono stati realizzati sfruttando il framework NUnit[], equivalente a JUnit ma su piattaforma .NET. Per ogni singola convenzione sono stati testati i diversi metodi di conversione dei nomi fornendo in input un nome e verificando che sia restituito modificato in accordo con la convenzione. Tutti i test hanno avuto esito positivo.

I test sui predicati invece sono stati effettuati utilizzato il framework FIT. Sono stati predisposti tre file HTML (*testConventions, testWithoutConventions* e *testWithConventions*) contenenti rispettivamente i test sul caricamento e scaricamento delle convenzioni e i test di utilizzo nelle due modalità. Anche in questo caso tutti i test hanno avuto esito positivo.



*Figura 14 – Test OOLibrary e convenzioni*

Di seguito vengono mostrati alcuni esempi di utilizzo della OOLibrary per mostrare il grado di interoperabilità raggiunto e il ruolo svolto dalle convenzioni.

Per gli esempi è stata utilizzata la classe Student (Figura 9, sezione 5.1), riscritta nei diversi linguaggi seguendo la relativa convenzione.

**Esempio oggetto Visual Basic .NET**

testWithoutConvention :-

new\_object('CStudent.Student',[311471,'ale','monta'], Obj),

Obj <- 'PrintStudent' returns Studente,

Obj <- get\_Id returns Matricola,

class('CStudent.Student,CStudent') <- get\_StaticProperty returns Value,

new\_object(' CStudent.Student, CStudent[]',[10], Array).

testWithConvention :-

load\_convention('VBConvention.dll','VBConvention.VBDotNet',Conv),

new\_object(Conv,'VBStudent.student, VBStudent',[311471,'ale','monta'], Obj),

Obj <- printStudent returns Student,

Obj.id <- get(Matricola),

class('VBStudent.Student, VBStudent').staticProperty <- get(Value),

new\_object(Conv, 'VBStudent.Student, VBStudent()',[10], Array).

Nel primo esempio non sono utilizzate le convenzioni. Si nota come la sintassi sia pesante dato che occorre indicare i nomi dei metodi tra apici singoli. Soprattutto è da sottolineare il fatto che per accedere ad una proprietà è necessario conoscere il nome del metodo associato a quest’ultima (get\_Id). Inoltre nel caso specifico di Visual Basic per creare un array occorre utilizzare i caratteri “[]” mentre il linguaggio prevede l’utilizzo di “()”.

Nel secondo esempio invece sono utilizzate le convenzioni ed è possibile notare come l’utilizzo delle proprietà sia più naturale (Oggetto.Proprietà) e la sintassi per creare un array rispetti la convenzione del linguaggio. Inoltre utilizzando le convenzioni non è più necessario adottare gli apici singoli per indicare i nomi dei metodi: il nome può essere indicato con l’iniziale minuscola, sarà la convenzione ad occuparsi di modificarlo opportunamente.

**Esempio oggetto Java**

Per utilizzare un oggetto Java è possibile tradurlo in .NET con IKVM.NET oppure inserire il file *.class* nella stessa directory in cui viene eseguita l’applicazione. In tal caso, al momento dell’utilizzo, la classe Java verrà caricata e tradotta dinamicamente in CIL. Questa funzionalità è resa possibile dal ClassPathAssemblyClassLoader messo a disposizione da IKVM.NET (sezione 1.5.3).

testWithoutConvention :-

new\_object('javastudent.Student',[311471,'ale','monta'], Obj),

Obj <- printStudent returns Studente,

Obj <- getId returns Matricola,

class('javastudent.Student') <- printInfoUni returns Uni,

new\_object('javastudent.Student[]',[10], Array).

testWithConvention :-

load\_convention('JavaConvention.dll','JavaConvention.Java',Conv),

new\_object('javastudent.student',[311471,'ale','monta'], Obj),

Obj <- 'PrintStudent' returns Studente,

Obj <- getId returns Matricola,

class('javastudent.Student') <- printInfoUni returns Uni,

new\_object('javastudent.Student[]',[10], Array).

Nel caso di utilizzo di classi Java, siccome il compilatore non introduce modifiche, come avviene per le proprietà in .NET, la convenzione potrebbe anche non essere utilizzata. Il vantaggio marginale che si ottiene è la possibilità di scrivere i metodi con la prima lettera maiuscola (ad esempio in caso di errore), infatti, in questo caso, se si utilizza la convenzione, quest’ultima modificherà il nome opportunamente e sarà quindi possibile invocare il metodo ('PrintStudent').

**Esempio concreto**

test(TotExams) :-

load\_convention('CSharpConvention.dll','CSharpConvention.CSharp',CSConv),

load\_convention('FSharpConvention.dll','FSharpConvention.FSharp',FSConv),

load\_convention('VBConvention.dll','VBConvention.VBDotNet',VBConv),

load\_convention('JavaConvention.dll','JavaConvention.Java',JaConv),

new\_object(CSConv, 'CStudent.Student, CStudent',[122345,'Ale',''], StudCS),

new\_object(FSConv, 'FStudent.Student, FStudent',[525718,'Mario',''], StudFs),

new\_object(VBConv, 'VBStudent.Student, VBStudent',[98765,'Gino',''], StudVB),

new\_object(JaConv, 'javastudent.Student',[476328,'Pippo',''], StudJa),

StudCS.exams <- get(Ex1),

StudFs.exams <- get(Ex2),

StudVB.exams <- get(Ex3),

StudJa <- getExams returns Ex4,

TotExams is Ex1 + Ex2 + Ex3 + Ex4.

Questo esempio ha lo scopo di mostrare come sia possibile, da Prolog, istanziare ed utilizzare oggetti scritti in diversi linguaggi. Inoltre si osserva che i tipi non strutturati di Prolog (stringhe, numeri, ecc.) sono interoperabili con i tipi primitivi dei linguaggi considerati. Infatti i valori contenuti nelle variabili Ex1, Ex2, Ex3 ed Ex4 possono essere sommati direttamente, senza ulteriori conversioni.

# Appendice

## Appendice A – Esempi di utilizzo di IKVM.NET

Di seguito sono riportati alcuni esempi di utilizzo di IKVM.NET nelle due modalità:

* **modalità dinamica**:

Ipotizzando di avere il file *JavaApp.jar* nella cartella corrente che necessiti della libreria *lib1.jar* nella sottodirectory *lib* . Per avviare l’applicazione sulla piattaforma .NET:

*> ikvm –jar –cp .\lib\lib1.jar JavaApp.jar*

* **modalità statica**:

Creazione di un singolo assembly: con estensione *.exe* se il jar è eseguibile oppure con estensione *.dll* se il jar è una libreria non eseguibile:

*> ikvmc JavaApp.jar*

Creazione di un singolo assembly specificando come class loader il ClassPathAssemblyClassLoader:

*> ikvmc -classloader:ikvm.runtime.ClassPathAssemblyClassLoader JavaApp.jar*

Creazione di un singolo assembly includendo tutti i file *.class* presenti nella directory bin (e sottodirectory) e il file *mylib.jar.* Viene specificato che l’assembly deve essere eseguibile:

*> ikvmc –assembly:App –target:exe –main:org.inc.Main –recurse:.\bin\\*.class lib\mylib.jar*

Creazione di un’applicazione che utilizza una libreria esterna:

*> ikvmc mylib.jar ;viene generato il file mylib.dll*

*> ikvmc –assembly:App –target:exe –main:org.inc.Main –recurse.\bin\\*.class –r:mylib.dll*

Creazione file jar contenenti gli stub delle classi .NET per poter scrivere applicazioni .NET in Java:

*> ikvmstub mscorlib.dll*

## Appendice B – Esempio di applicazione .NET scritta in Java con IKVM.NET

Di seguito è riportato un esempio di applicazione scritta in C# e Java sfruttando IKVM.NET per mostrare le conversioni sintattiche introdotte da *ikvmstub.exe*.

**Applicazione di esempio scritta in C#**

using System;

using System.Xml;

using AttributesLib;

namespace Test

{

[Author("Alessandro")]

class Program

{

static void Main(string[] args)

{

XmlDocument aDoc = new XmlDocument();

aDoc.NodeInserted += new XmlNodeChangedEventHandler(NodeInserted);

XmlNode aNode = aDoc.CreateElement("Element");

aDoc.AppendChild(aNode);

aDoc.Save("XmlDoc.xml");

aDoc.NodeInserted -= NodeInserted;

Console.WriteLine(XmlNodeType.Element);

}

static void NodeInserted(object sender, XmlNodeChangedEventArgs e)

{

Console.WriteLine("Inserito nuovo nodo XML: " + e.Node.Name);

}

}

}

**Applicazione di esempio scritta in Java**

package test;

import cli.System.\*;

import cli.System.Xml.\*;

import cli.AttributesLib.\*;

@AuthorAttribute.Annotation("Alessandro")

public class Program implements XmlNodeChangedEventHandler.Method

{

public static void main(java.lang.String[] args)

{

XmlDocument aDoc = new XmlDocument();

XmlNodeChangedEventHandler handler =

new XmlNodeChangedEventHandler(new Program());

aDoc.add\_NodeInserted(handler);

XmlNode aNode = aDoc.CreateElement("Element");

aDoc.AppendChild(aNode);

aDoc.Save("XmlDoc.xml");

aDoc.remove\_NodeInserted(handler);

Console.WriteLine(XmlNodeType.Element);

}

public void Invoke(java.lang.Object sender, XmlNodeChangedEventArgs e)

{

Console.WriteLine("Inserito nuovo nodo: " + e.get\_Node().get\_Name());

}

}

## Appendice C – Esempi di utilizzo di Ja.NET

Di seguito sono riportati alcuni esempi di utilizzo di Ja.NET:

Compilazione di tutti i sorgenti presenti nella directory corrente:

*> javac \*.java*

Creazione di un singolo assembly contenente tutte le classi presenti nella directory corrente:

*> bam –cf Lib.dll \*.class*

Esecuzione di un’applicazione su piattaforma .NET:

*> java –cp App.dll org.inc.Main*

Compilazione di tutti i sorgenti presenti nella directory corrente con riferimento ad una libreria esterna già compilata. Viene specificato di compilare i sorgenti sia per la piattaforma Java che per quella .NET:

*> javac –cp Lib.dll –both \*.java*

## Appendice D – Esempio di applicazione .NET scritta in Java con Ja.NET

Di seguito è riportato un esempio di applicazione scritta in Java sfruttando Ja.NET per mostrare gli adattamenti necessari per utilizzare le caratteristiche del CIL.

Per l’applicazione di esempio scritta in C# fare riferimento all’appendice B.

**Applicazione di esempio scritta in Java**

package test;

import System.\*;

import System.Xml.\*;

import AttributesLib.AuthorAttribute;

@AuthorAttribute(Author = "Alessandro")

public class Program

{

public static void main(java.lang.String[] args)

{

XmlDocument aDoc = new XmlDocument();

XmlNodeChangedEventHandler handler =

new XmlNodeChangedEventHandler(NodeInserted);

aDoc.add\_NodeInserted(handler);

XmlNode aNode = aDoc.CreateElement("Element");

aDoc.AppendChild(aNode);

aDoc.Save("XmlDoc.xml");

aDoc.remove\_NodeInserted(handler);

Console.WriteLine(XmlNodeType.Element);

}

public static void NodeInserted(System.Object sender, XmlNodeChangedEventArgs e)

{

Console.WriteLine("Inserito nuovo nodo: " + e.get\_Node().get\_Name());

}

}

## Appendice E – Approfondimento *NETFileRunner.exe*

*NETFileRunner.exe* è un’applicazione che ho scritto per superare i problemi derivanti dal sistema di class loading implementato da IKVM.NET. In particolare i problemi riguardano il caricamento delle fixture per eseguire i test FIT. Nelle tabelle HTML, infatti, il nome della classe che implementa la fixture è specificato con la notazione Java, “elencoPackage.NomeClasse”. Per fare in modo che il motore FIT tradotto con IKVM.NET sia in grado di caricare le classi specificate con questo nome si è deciso di tradurre il motore indicando come class loader l’*AppDomainAssemblyClassLoader*. Ciò permette al motore di caricare, con il nome nel formato Java, tutte le classi presenti nell’AppDomain corrente.

*NETFileRunner.exe* pertanto svolge i seguenti passi:

1. carica, nell’AppDomain corrente, l’assembly contenente le fixtures;
2. lancia l’esecuzione dei test, utilizzando la versione di FIT tradotta con il class loader citato precedentemente.

Il comando per eseguire *NETFileRunner.exe* è il seguente:

*NETFileRunner input-file output-file fixture-lib-file*

I parametric rappresentano:

* **input-file**: file HTML che contiene la definizione dei test;
* **output-file**: file HTML in cui scrivere i risultati dei test;
* **fixture**-**lib-file**: assembly contenente le fixtures necessarie per i test.

In tal modo si è evitato di cablare le fixtures direttamente all’interno del motore FIT (soluzione singolo assembly) ottenendo di fatto la riutilizzabilità dello stesso. Infatti, è possibile eseguire test diversi, con fixtures diverse specificando, nell’invocazione di *NETFileRunner.exe*,un assembly differente, il tutto senza modificare o ritradurre il motore FIT.

## Appendice F – Esempio di utilizzo di tuProlog su piattaforma .NET

In questa appendice viene mostrato il procedimento passo-passo per l’utilizzo della versione .NET dell’interprete tuProlog in un’applicazione.

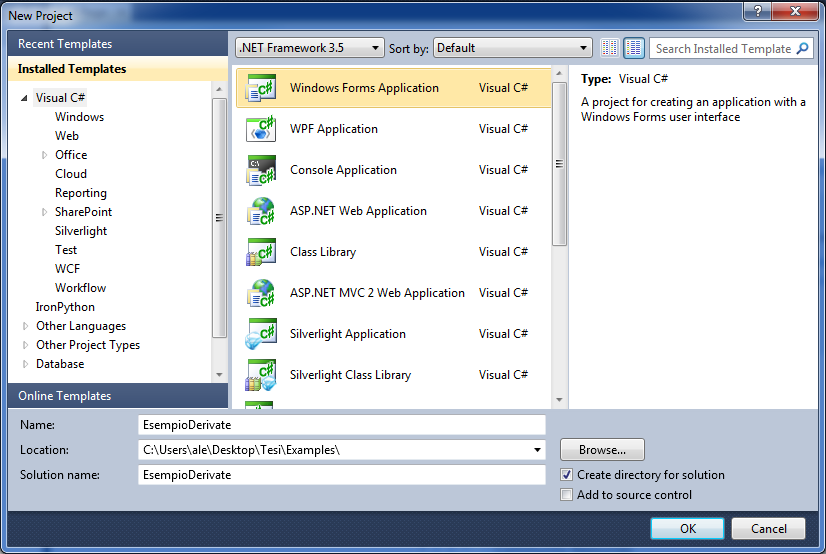
In particolare è stata implementata una semplice applicazione per la derivazione simbolica di funzioni.

L’ambiente di sviluppo utilizzato in questa guida è Microsoft Visual Studio 2010, ma le stesse operazioni possono essere realizzate in modo simile tramite altri ambienti di sviluppo per .NET.

Il linguaggio di programmazione può essere uno qualsiasi dei linguaggi supportati dall’ambiente .NET. Nello specifico è stato utilizzato il linguaggio C#.

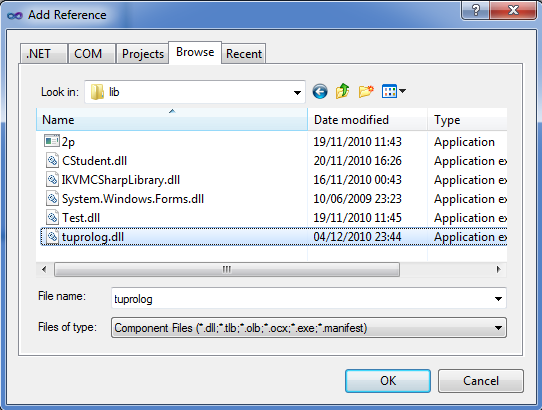
Operazioni:

1. aprire lo strumento di sviluppo;
2. selezionare nel menù *File -> New -> Project;*
3. nella lista a sinistra selezionare *Visual C#*. Nell’elenco dei *Template* selezionare il tipo di applicazione da creare: in questo caso *Windows Forms Application*. Inserire il nome dell’applicazione, la posizione nel filesystem e confermare (Figura F.1);



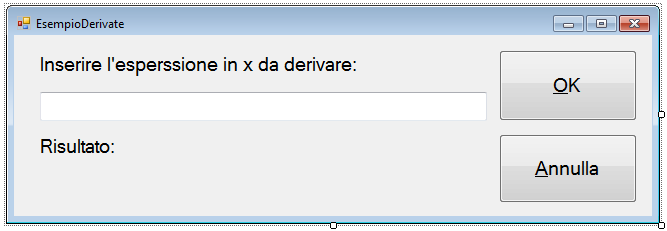
*Figura F.1 – Nuovo progetto*

1. per poter utilizzare il motore di tuProlog è necessario inserire il riferimento all’assembly che lo contiene (*tuprolog.dll*). Cliccare con il tasto destro su *References*,nel pannello *Solution Explorer*, e successivamente su *Add References…* A questo punto nella tab *Browse* navigare nel filesystem fino a localizzare l’assembly, selezionarlo e confermare (Figura F.2);



*Figura F.2 – Aggiunta riferimento a tuProlog*

1. inserire il riferimento all’assembly *IKVM.OpenJdk.Core.dll* che contiene i package Java principali. Utilizzare le stesse modalità del punto 4;
2. disegnare l’interfaccia desiderata (Figura F.3);

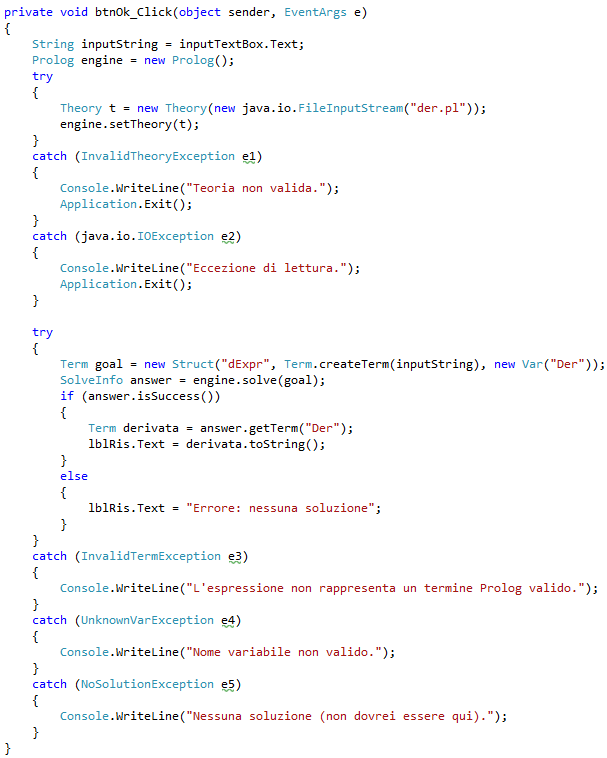


*Figura F.3 – Interfaccia applicazione*

1. nel codice inserire lo using del namespace necessario per utilizzare tuProlog:

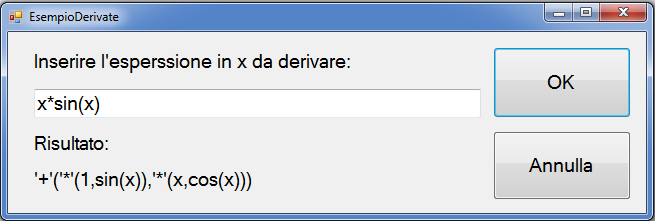
using alice.tuprolog;

1. scrivere il codice di implementazione per l’handler del bottone Ok (Figura F.4);



*Figura F.4 –* Implementazione *handler bottone Ok*

1. eseguire l’applicazione premendo il tasto F5 (Figura F.5).

**

*Figura F.5 – Esempio di utilizzo*

Come si nota dalla Figura F.4 all’interno del codice C# sono stati utilizzati alcuni tipi Java (java.io.FileInputStream e java.io.IOException). Ciò è possibile perché è stata importata la libreria di IKVM.NET (*IKVM.OpenJDK.Core.dll*) che contiene l’implementazione in .NET dei principali package Java (java.lang, java.util, java.io, …). Questo significa che quel particolare assembly contiene esattamente tutte le classi (con gli stessi metodi) presenti nei principali package Java, con la differenza che sono a tutti gli effetti classi .NET. Pertanto possono essere utilizzati da un qualunque linguaggio .NET. Discorso analogo vale per le classi del motore tuProlog: *tuprolog.dll* contiene le stesse classi della distribuzione Java ma implementate in .NET. Queste classi sono generate dal traduttore fornito con IKVM.NET (sezione 1.5).

# Bibliografia

1. **IKVM.NET**

<http://www.ikvm.net/>

1. **Java Virtual Machine Specification**

<http://java.sun.com/docs/books/jvms/second_edition/html/VMSpecTOC.doc.html>

1. **Piattaforma .NET**

<http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-335.pdf>

1. **Open JDK 6**

<http://openjdk.java.net/projects/jdk6/>

1. **Java bytecode**

<http://www.ibm.com/developerworks/ibm/library/it-haggar_bytecode/>

1. **Common Intermediate Language (CIL)**

<http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-335.pdf>

1. **Microsoft .NET Framework**

<http://www.microsoft.com/net/>

1. **Mono .NET Framework**

<http://www.mono-project.com/Main_Page>

1. **Licenza zlib**

<http://www.gzip.org/zlib/zlib_license.html>

1. **Ja.NET**

<http://www.janetdev.org/>

1. **Apache Harmony JVM**

<http://harmony.apache.org/index.html>

1. **Eclipse Public License - v 1.0**

<http://www.eclipse.org/legal/epl-v10.html>

1. **Visual C++ 2008 runtime libraries**

<http://www.microsoft.com/downloads/en/details.aspx?FamilyID=9b2da534-3e03-4391-8a4d-074b9f2bc1bf&displaylang=en>

1. **tuProlog**

<http://alice.unibo.it/xwiki/bin/view/Tuprolog/>

1. **tuProlog: A Light-Weight Prolog for Internet Applications and Infrastructures**

<http://alice.unibo.it/xwiki/bin/view/Publications/TuprologPadl01>

1. **Licenza LGPL**

<http://www.gnu.org/licenses/lgpl.html>

1. **Multi-paradigm Java–Prolog integration in tuProlog**

<http://alice.unibo.it/xwiki/bin/view/Publications/TuprologScico57>

1. **Framework P@J – Integrating Java and Prolog through Generic Methods and Type Inference**

<http://apice.unibo.it/xwiki/bin/view/Publications/PatjSac08>

1. **Progetto e sviluppo del supporto all'interoperabilità fra l'interprete tuProlog e linguaggi su piattaforma Microsoft.NET – Marco Albertin**

<http://www.alice.unibo.it/xwiki/bin/view/Theses/AlbertinTuprolognet08>

1. **JUnit 4.8.2**

<http://www.junit.org/>

1. **Fit – Framework for Integrated Test**

<http://fit.c2.com/>

1. **GUIRunner**

<http://martin.busik.de/fit-guirunner.html>

1. **Test Prestazioni Prolog**

<http://www.sics.se/isl/sicstuswww/site/sicstus4.html>

1. **NUnit**

<http://www.nunit.org/>