UNIVERSITÀ DEGLI STUDI
DI MODENA E REGGIO EMILIA

Laurea in Informatica

Analisi e Valutazione di Integrazione tra Linguaggi di Programmazione Differenti in un Contesto Aziendale

Francesco Picchietti

Tesi di Laurea

Relatori:

Prof. Giacomo Cabri

Prof. Riccardo Martoglia

Anno Accademico 2020/2021

RINGRAZIAMENTI

Ringrazio innanzitutto i professori Giacomo Cabri e Riccardo Martoglia per avermi pazientemente aiutato durante il periodo di tirocinio e la scrittura della tesi.

Ringrazio la mia famiglia, per aver sempre creduto in me ed avermi supportato in ogni momento nonostante la mia ansia costante. Ringrazio anche mio fratello Marco per avermi sempre fatto sorridere e tranquillizzare chiedendomi “T’han boccià?” dopo ogni esame.

Un supporto grandissimo mi è sempre stato dato dalla mia fidanzata, Beatrice. Ogni volta che mi sono sentito demoralizzato è sempre stata la mia àncora di salvezza.

Un grazie anche ai miei coinquilini di Modena e ai miei amici, sempre disponibili e sempre vicini in qualsiasi momento.

Un ringraziamento particolare va a due professori delle superiori che mi hanno sconsigliato di intraprendere qualsiasi percorso scientifico. Grazie per aver creduto in me.

Se sono qui ora lo devo a tutti voi, grazie.

PAROLE CHIAVE

Integrazione

Compilazione

IronPython

C#

Python

Indice

Elenco delle figure

Introduzione

La maggior parte degli sviluppatori software si concentra sull'utilizzo di un singolo linguaggio di programmazione, cercando di risolvere qualsiasi problema in modi spesso complicati. Il seguente studio, testato in ambito aziendale, ha lo scopo di valutare l'utilizzo di un processo di integrazione fra linguaggi di programmazione differenti: un linguaggio di programmazione esclusivamente ad oggetti come C#, e Python, il quale si presta bene alla creazione di script.

Questo studio nasce dalla necessità di automatizzare determinati processi di assistenza e analisi su macchine in remoto tramite linguaggio Python, da poter in un secondo momento integrare all'interno di un sistema più ampio scritto in C#.

Nella presente tesi è stato studiato il meccanismo di integrazione tra linguaggi di programmazione, valutandone attentamente i punti di forza, le debolezze e confrontandolo con altri processi simili.

Lo studio è suddiviso in due parti. Nella prima parte viene presentato il caso di studio. Viene descritto il contesto rispetto al quale i software sono stati sviluppati e vengono elencati i motivi per cui si è deciso di adottare un processo di integrazione. Successivamente sono presentati i due linguaggi, mettendone a confronto i punti di forza e i loro principali impieghi. Si arriva poi all'argomento principale della tesi: l'integrazione. Qui vengono descritti i due metodi principali per poter integrare due linguaggi di programmazione: ovvero tramite un software, IronPython, oppure tramite un processo di compilazione. Infine, all'interno di questo capitolo, vengono messi a confronto a livello teorico i due meccanismi.

La seconda parte mette in pratica le premesse che sono state fatte precedentemente tramite lo sviluppo di due software per l'assistenza. Il primo software si occupa di raccogliere informazioni sulla macchina, al fine di poterne valutare a posteriori la compatibilità con software sviluppati dall'azienda. Il secondo serve per effettuare

operazioni su specifici database Firebird. Questo programma viene utilizzato nel momento in cui alcune macchine subiscano rallentamenti, rendendo quindi necessario effettuare un processo detto *shrinking*.

Alla fine di questo capitolo vi è uno studio relativo ai risultati ottenuti e alle misurazioni delle prestazioni dei software. Le considerazioni fatte alla fine di questa parte non sono basate solamente sulla correttezza dei risultati ottenuti dai programmi ma verranno anche analizzati il livello di efficacia dell'integrazione e la velocità di esecuzione.

Parte I

Il Caso di studio

Capitolo 1

Presentazione del contesto e dei linguaggi coinvolti

1.1 Il contesto aziendale

Lo studio e l'utilizzo di sistemi di integrazione sono stati fatti all'interno di un contesto aziendale durante il periodo di tirocinio.

L'azienda ospitante si occupa della progettazione e della creazione di macchinari per la stampa di prodotti ceramici dette stampanti. Le occupazioni primarie dell'ufficio informatico sono lo sviluppo del software per l'interazione con componenti della macchina e l'assistenza alle stampanti acquistate dalle aziende.

Due principali processi di assistenza si prestavano bene ad un processo di automazione in quanto ripetitivi e spesso da dover effettuare su più macchine contemporaneamente. Il primo software, sviluppato in linguaggio Python, si occupa della raccolta di informazioni riguardanti la macchina. Una volta terminata l'esecuzione del programma, i risultati vengono analizzati al fine di poter valutare la compatibilità tra diverse versioni di software e del sistema operativo ospitante. Il secondo, scritto sempre in Python, si occupa della pulizia di un database utilizzato dal software che gestisce la macchina di stampa. Il processo richiedeva dunque l'interazione con un database Firebird. Per poter effettuare questo tipo di operazioni si è reso necessario sfruttare delle specifiche API per potersi interfacciare con il database in questione.

Entrambi i software dovevano essere integrati all'interno di un programma per l'automazione delle assistenze scritto in C#. Lo sviluppo di queste due estensioni in un linguaggio differente nasce dal fatto che Python si presta bene alla manipolazione di stringhe e anche all'interazione con i database.

1.2 Python

“Python può essere considerato come un linguaggio di programmazione di più "alto livello" rispetto alla maggior parte degli altri linguaggi, esso è orientato a oggetti, il che gli consente, tra gli altri usi, a sviluppare applicazioni complesse, scripting, applicazioni web computazione numerica e system testing [1]”.

I linguaggi di scripting come Python sono più produttivi dei linguaggi convenzionali, come C e Java, per problemi che includono la manipolazione di stringhe e la ricerca nei dizionari.

Tra le peculiarità di Python rientra il fatto di essere un linguaggio multi-paradigma, ovvero il supportare il paradigma *Object-oriented*, la *programmazione strutturale* e la *programmazione funzionale*.

Le caratteristiche riconoscibili a primo impatto sono il fatto di non avere una tipizzazione statica; quindi, non è necessario definire a priori il tipo delle variabili. In secondo luogo, non vengono utilizzati punto e virgola e parentesi per poter definire sezioni di codice, al loro posto viene utilizzata in maniera molto scrupolosa l’indentazione.

Python viene considerato un linguaggio di scripting, ma, nonostante sia molto utile a questo scopo, l’enorme quantità di librerie a sua disposizione ne consente un utilizzo molto più strutturato e adatto a sviluppare applicazioni complesse.

Una caratteristica del linguaggio che sarà di importanza fondamentale per le tipologie di integrazione è il fatto di essere un linguaggio interpretato o *pseudocompilato*. “Python è un linguaggio pseudocompilato: un interprete si occupa di analizzare il codice sorgente, ovvero file testuali con estensione .py, e, se sintatticamente corretto, di eseguirlo. Non esiste quindi una fase di compilazione separata, come avviene in C, che generi un file eseguibile partendo dal sorgente. [2]”

1.2.1 Impieghi del linguaggio

I principali impieghi di Python si trovano in ambito di *applicazioni web*, *computazione scientifica*, *intelligenza artificiale* e per il *data mining*

Per quanto riguarda le applicazioni web, Python può essere sia usato come linguaggio di scripting, oppure tramite i web framework come *Django*, *Flask* e *Pyramid*. Tramite il loro utilizzo i programmatori sono agevolati nello sviluppo e nel mantenimento di applicazioni complesse. Il framework *Twisted* viene utilizzato per la comunicazione fra computer ed è utilizzato anche da *Dropbox*.

In campo scientifico, il linguaggio Python viene largamente utilizzato. La computazione scientifica viene resa efficiente tramite l'utilizzo di librerie come *NumPy*, *SciPy* e *Manplotlib*. Per la biologia e bio-informatica vengono utilizzate librerie come *BioPython;* per la visione artificiale viene utilizzato *OpenCV*.

Oggi Python può essere sfruttato per l'intelligenza artificiale perché è un linguaggio di scrittura veloce e open source, vantaggi che lo rendono perfetto per l'AI. Permette di tradurre concetti particolarmente complessi in un linguaggio che più di ogni altro si avvicina allo pseudocodice. Inoltre, esistono librerie come *Keras* e *TensorFlow*, le quali contengono molte informazioni sulle funzionalità dell'apprendimento automatico.

L'uso di Python è molto diffuso nell'analisi dei dati e nell'estrazione delle informazioni utili per le aziende, per questo viene utilizzato nelle aziende di Big Data. Questo perché, oltre alla sua semplicità, ha il grande vantaggio di possedere librerie di elaborazione dati come *Pydoop*, le quali danno un notevole aiuto per quanto riguarda l'elaborazione di dati. Altre librerie come *Dask* e *Pyspark* semplificano ulteriormente l'analisi e la gestione degli stessi.

1.3 C#

C# è un linguaggio multi-paradigma ma che si presta principalmente a quella che è detta programmazione a oggetti. Esso è stato sviluppato da Microsoft e si appoggia al framework .NET. La sintassi di C# prende ispirazione da linguaggi come *C++*, *Java* e *Delphi*.

Per quanto riguarda quella che sembra essere una grandissima somiglianza con Java, da parte di questo linguaggio, "Microsoft afferma che la struttura di C# è più vicina a quella del linguaggio C++ piuttosto che a Java [3]".

La sintassi del linguaggio presenta diverse specifiche, tra cui, nomi delle variabili *case-sensitive*, ovvero che risentono dell'utilizzo di maiuscole o minuscole. Ogni specifica deve essere chiusa, a differenza di Python, con un punto e virgola e per raggruppare le specifiche vengono utilizzate le parentesi graffe. Seguendo le convenzioni dei linguaggi a oggetti, le specifiche sono raggruppate in *metodi*, che a loro volta sono raggruppati in *Classi*. Le *Classi* sono raggruppate nei *Namespace*. L'ereditarietà è singola ma non vi è limite al numero delle *Interfacce* che possono essere ereditate.

1.3.1 Caratteristiche principali

Esistono alcune differenze con i linguaggi di programmazione C e C++ la prima differenza riguarda l'utilizzo dei puntatori. I puntatori in C++ possono essere utilizzati in ogni parte del codice mentre in C#, per preservare possibili errori dovuti a un'errata gestione dell'aritmetica dei puntatori, devono essere inseriti in particolari blocchi del codice marcati come unsafe. Sempre per evitare eventuali problemi come *dangling reference* o *memory leak*, la gestione della memoria non è a carico del programmatore ma viene curata dal *garbage-collector*, il quale si occupa di

deallocare dinamicamente gli oggetti quando non esiste più alcun riferimento ad essi. Esistono inoltre controlli automatici per eventuali *overflow*.

Le differenze con Java sono, innanzitutto, che C# utilizza una metodologia di commenti che prevedono l'utilizzo della sintassi XML. Quello che in Java è chiamato *package,* in C# viene chiamato *namespace*. Viene infine utilizzato un diverso modo per richiamare i metodi costruttori di una classe.

1.3.2 Compilazione

Non è possibile definire in maniera statica se il linguaggio C# sia un linguaggio compilato o interpretato. Essendo legato al framework .NET, il codice viene compilato tramite i criteri della Just In Time, ovvero una compilazione effettuata durante l'esecuzione del programma e non prima. In un primo momento il codice sorgente viene convertito dal framework in un codice intermedio detto *CIL* e successivamente, durante l'esecuzione vera e propria, il Common Language Runtime converte il codice CIL, man mano che il codice viene eseguito, in linguaggio macchina specifico per l'hardware ospite. Questo passaggio rende la velocità di esecuzione più lenta nel primo momento in cui il codice viene eseguito ma, ogni volta che l'esecuzione viene ripetuta, essa diventa sempre più veloce, diventato ottimale dopo un certo numero di esecuzioni.

Un secondo tipo di compilazione è quello che viene detto *compilazione Ngen,* il quale permette di convertire il codice CIL in codice macchina una sola volta e prima dell'esecuzione.

1.3.3 Impieghi del linguaggio

Essendo C# un linguaggio general-purpose, risulta essere uno dei più utilizzati e uno dei più versatili. Esso, quindi, ha vari campi di utilizzo che spaziano dai siti web, passando per le app desktop e arrivando ai videogiochi. Ci sono però tre aree in cui C# è particolarmente sfruttato.

Il primo impiego di C# è lo sviluppo web. Tramite il framework .NET possono essere creati siti web dinamici. Questo linguaggio, essendo pensato primariamente per essere ad oggetti, rende più semplice lo sviluppo di applicazioni web scalabili e facilmente mantenibili.

Il secondo è quello di creare applicazioni Windows. I software scritti con questo linguaggio di programmazione hanno bisogno del framework .NET per funzionare al meglio; quindi, il caso d'uso per questo linguaggio è lo sviluppo di programmi che siano specifici per le architetture Microsoft.

Il terzo impiego è quello di essere uno dei linguaggi più utilizzati per sviluppare videogiochi. Il linguaggio C# interagisce direttamente con Unity, ovvero un motore grafico utilizzato per sviluppare videogiochi sia 2D che 3D.

1.4 Differenze tra i due linguaggi

Python è una piattaforma con codice sorgente aperto, mentre C# è sviluppata da Microsoft, quindi presenta alcune limitazioni. Python è un linguaggio multi-paradigma, mentre C# supporta solo la programmazione OOP, ovvero orientata agli oggetti, e richiede .NET Framework. Il linguaggio C# è digitato staticamente mentre Python dinamicamente. C#, inoltre, offre delle prestazioni superiori quando si tratta di motori di gioco, esso è infatti particolarmente utile per la creazione di applicazioni e giochi desktop di Windows. Python non supporta puntatori, a differenza di C#.

In questo caso, quando si parla di velocità, non viene intesa la velocità di esecuzione ma la velocità di programmazione. In primo luogo, Python ha una sintassi molto più semplice di C#. D'altro canto, C# è più familiare, se si conosce Java o C allora utilizzare C# risulta più immediato. Mentre C# è tipizzato staticamente, Python è tipizzato dinamicamente; quindi, i tipi delle variabili non devono essere rigorosamente assegnati durante la scrittura del software. Questa tipizzazione può essere interpretata in due modi differenti: la velocità e semplicità di scrittura di codice risulta migliore, oppure il codice risulta meno chiaro in quanto non si conosce la natura delle variabili che vengono utilizzate. Python, durante lo sviluppo di software, è anche più veloce nel momento in cui si deve eseguire il codice, in quanto, non avendo il passaggio di compilazione, esso viene eseguito immediatamente. Nonostante il tempo di compilazione non sia per forza lungo, può richiedere più tempo per il testing e il debugging.

C# utilizza, come tanti altri linguaggi, delle parentesi annidate, graffe e tonde. Python, invece, utilizza l'indentazione per poter distinguere parti di codice e sequenze di esecuzione. Anche in questo caso possono esserci interpretazioni opposte: il codice può essere visto come più pulito e leggibile, oppure di difficile comprensione.

Per quanto riguarda le performance c'è una grande differenza fra C# e Python. Il primo è un linguaggio compilato mentre il secondo è un linguaggio interpretato. La velocità di Python dipende quindi pesantemente dall'interprete, gli interpreti maggiormente utilizzati sono *CPython* e *PyPy*. Il linguaggio prestazionalmente più performante è C#.

Le applicazioni di entrambi i linguaggi, general purpose, sono varie. Essi possono essere usati per qualsiasi cosa, dallo sviluppo di videogiochi al machine learning, ma non sono uguali su tutti i fronti. Il machine learning è appunto un esempio di come i due linguaggi siano differenti. Python, sotto questo punto di vista, è migliore, in quanto possiede un enorme numero di software specifici: *Numpy*, *SciPy*, *TensorFlow*,

PyTorch, *Apache Spark*, *Keras* e altri. Per chi sviluppa in Python, considerato il miglior linguaggio per il machine learning, sono disponibili un gran numero di tutorial o documentazione per poter sfruttare al meglio queste potenzialità. Per quanto riguarda il machine learning in C#, vengono utilizzate librerie come *Accord.NET* oppure ML.NET e dei collegamenti a *TensorFlow*. Il *Microsoft Cognitive Toolkit (CNTK)* ha un supporto sia per Python che per C#.

C# ha una struttura più organizzata, essendo un linguaggio a oggetti non ci sono inconsistenze nelle regole o nella sintassi. Per questo motivo esso può risultare più lento da imparare e da utilizzare per programmare. C# può inoltre fare quasi tutto quello che fa Python e ha un tempo di esecuzione più breve. Python è più facile sia da imparare che da scrivere. Esso ha inoltre un maggior numero di librerie standard ed è più semplice da utilizzare per il machine learning. Entrambi i linguaggi sono gratuiti, con un set di tool maturi, comunità attive, un buon numero di framework e librerie. Hanno inoltre delle applicazioni estese in molti campi di programmazione.

Le differenze sono schematizzate in figura 1.

	Python	C#
Facile apprendimento	✔	✗
Open source	✔	✗
Tipizzato staticamente	✗	✔
Parentesi graffe e semicolon	✗	✔
Indentazione	✔	✗
Compilazione	✗	✔
Gestione puntatori	✗	✔
Multi ereditarietà	✔	✗
Multi paradigma	✔	✗

Figura 1 – Tabella differenze Python C#

Capitolo 2

Integrazione

2.1 Introduzione ad IronPython

IronPython è un software che si occupa di integrare codice Python all'interno del codice C# e viceversa. Tramite questo programma è possibile utilizzare librerie .NET all'interno di codice Python oppure di creare o utilizzare librerie Python all'interno di un software C#. IronPython è scritto interamente in C# ed è sostenuto dal *Dynamic Language Runtime*. Questo software è formato principalmente dall'engine IronPython, oltre che pochi altri tools per renderlo facilmente utilizzabile.

IronPython potrebbe essere una soluzione conveniente dal punto di vista di integrazione per i seguenti motivi.

I cicli di sviluppo, in Python, sono normalmente più rapidi rispetto a C#. Tramite l'utilizzo di un linguaggio dinamico, uno script può essere realizzato con meno codice, rendendo IronPython ideale per prototipi di applicazioni o sistemi di scripting nei quali altrimenti sarebbe necessario impiegare molto più tempo.

Esistono due tipi di programmatori per i quali IronPython potrebbe essere particolarmente utile. Il primo è il gruppo di programmatori Python, i quali, possono avere a disposizione un'implementazione di Python che gira su una piattaforma diversa a quella convenzionale. Il secondo tipo è quello dei programmatori .NET che potrebbero essere interessati alla possibilità che un linguaggio di scripting possa essere inserito in un'applicazione esistente.

Senza una fase di compilazione, programmare tramite l'utilizzo di IronPython può essere molto più veloce che utilizzando il linguaggio .NET, il che tipicamente richiede meno codice e risulta più leggibile. Un ciclo rapido di edit-run rende

IronPython ideale per la prototipazione e per l'utilizzo come linguaggio di scripting. Le classi possono essere tradotte, se necessario, in linguaggio C# con pochissimi cambiamenti per l'interfaccia di programmazione. Nonostante IronPython possa essere considerato un linguaggio nuovo, Python non lo è. Python è infatti un linguaggio maturo e stabile creato nel 1991 e in continuo sviluppo. La sintassi e i costrutti base sono stati utilizzati ampiamente durante tanti anni di esperienza di programmazione.

È risaputo di come Python sia sintatticamente più conciso di C#. Ma è anche vero che Python sia semanticamente più conciso, con costrutti del linguaggio che permettono di esprimere concetti complessi in poche righe di codice.

2.1.1 Il motore IronPython

IronPython utilizza il *Dynamic Language Runtime* (*DLR)*, ovvero un framework per scrivere linguaggi dinamici per .NET e che estende la *Common Language Runtime* (*CLR)*. Il fulcro del Framework .NET è appunto la CLR, che esegue programmi già compilati da codice sorgente in bytecode. Ogni linguaggio che può essere compilato in IL può essere eseguito da .NET.

I principali linguaggi .NET, ovvero VB.NET e C#, sono linguaggi tipizzati *staticamente*. Python, invece, appartiene a una differente classe di linguaggi, in quanto esso è tipizzato *dinamicamente*. Il motore IronPython compila il codice Python in codice IL, il quale viene eseguito dalla CLR. Opzionalmente, IronPython può compilare in *assembly*, che possono essere utilizzati per creare distribuzioni di applicazioni esclusivamente binarie.

Gli *assembly* consistono in librerie .NET o degli eseguibili. .NET è molto importante per gli *assembly*, in quanto in essi risiede il framework, sotto forma di librerie. A

causa della gestione della memoria e delle funzionalità di sicurezza che vengono garantite, il codice negli assembly .NET è chiamato codice gestito. Gli assembly contengono il codice compilato dal linguaggio .NET in un linguaggio bytecode intermedio (IL). Esso, infatti, gira sul compilatore Just-In-Time (JIT) per un'esecuzione rapida.

Nell'immagine in figura 2 si può notare come il codice Python sia compilato ed eseguito dal motore di IronPython.

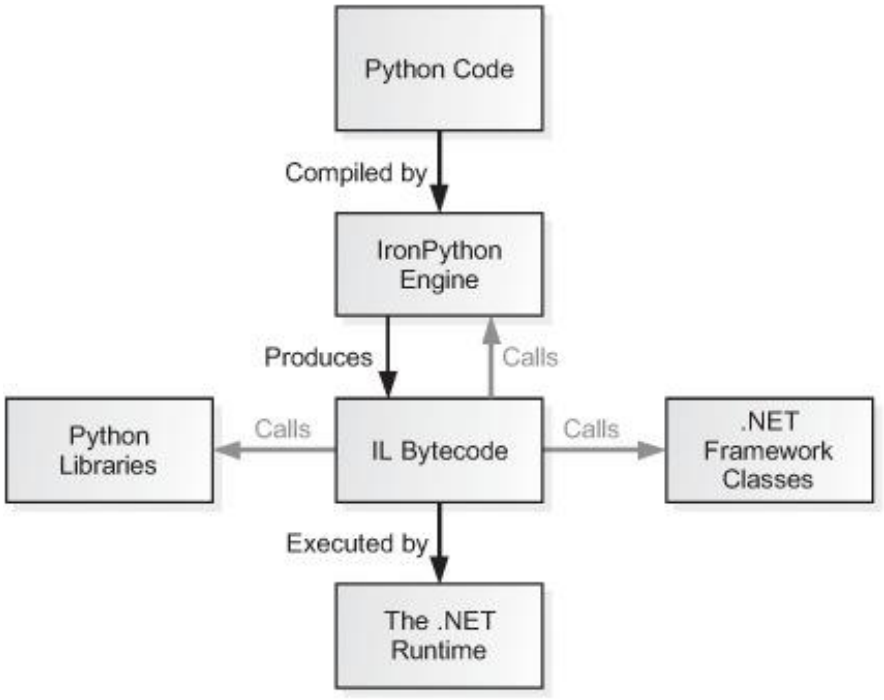

Figura 2 – Engine di IronPython

Questa figura mostra la prima versione in assoluto di IronPython, anche se nella versione successiva di IronPython, IronPython2, viene sfruttata la nuova tecnologia della *DLR*.

2.1.2 Common Language Runtime (CLR)

Il *Common Language Runtime (CLR)* è l'insieme composto dalla macchina virtuale e le librerie della piattaforma .NET. Ovvero l'ambiente di esecuzione del codice intermedio che i compilatori del framework .NET ottengono tramite una traduzione del codice di alto livello. Questo ambiente di esecuzione è utilizzato in ambiente Microsoft ma è possibile trovarne alcune implementazioni anche in sistemi Unix e Linux.

2.1.3 Dynamic Language Runtime (DLR)

Dynamic Language Runtime (DLR) è un ambiente di runtime che estende la *Common Language Runtime (CLR)* con un insieme di servizi per linguaggi dinamici. DLR semplifica lo sviluppo per questo tipo di linguaggi che devono eseguire nel framework .NET, aggiungendo ai linguaggi tipizzati in modo statico delle funzionalità dinamiche. Esso è utilizzato per implementare linguaggi dinamici come se fossero librerie appartenenti al framework .NET.

Esistono tre tipi di servizi fondamentali che il DLR mette a disposizione in aggiunta a quelli già disponibili del CLR.

Alberi di espressioni per rappresentare la semantica del linguaggio. Gli alberi delle espressioni rappresentano codice in una struttura dati simile a un albero dove ogni nodo è un'espressione. Questo tipo di struttura dati viene utilizzata allo stesso modo anche negli interpreti dei linguaggi dinamici.

Memorizzazione nella cache del sito della chiamata. Un sito di chiamata dinamica è un punto del codice in cui si esegue un'operazione su oggetti dinamici. DLR memorizza le informazioni riguardo a questa operazione e, nel caso in cui essa venga ripetuta, l'informazione viene restituita più rapidamente.

Interoperabilità con gli oggetti dinamici. Viene offerto un insieme di classi e interfacce che rappresentano operazioni e oggetti dinamici e che possono essere usate dagli autori delle librerie dinamiche.

La DLR utilizza i binder nei siti di chiamata per comunicare non solo con .NET Framework, ma anche con altri servizi e infrastrutture. I binder utilizzano gli alberi delle espressioni per rendere note la semantica di un linguaggio e la modalità di esecuzione delle operazioni in un sito di chiamata. Ciò consente ai linguaggi dinamici e a quelli tipizzati in modo statico che usano DLR di condividere le librerie e di accedere a tutte le tecnologie supportate. Nell'immagine in figura 3 si può notare come i binder della Dynamic Language Runtime rendano possibile la condivisione di librerie.

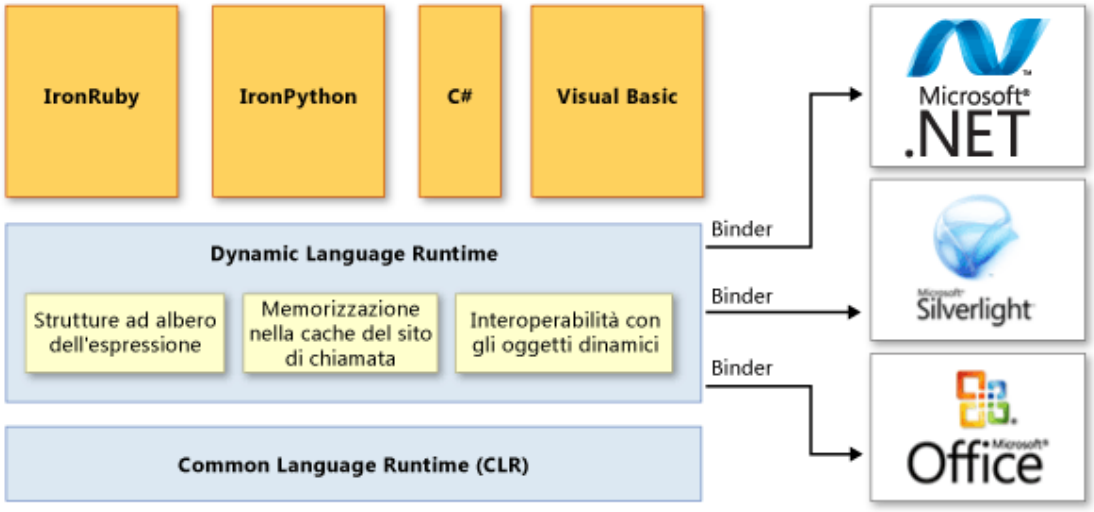

Figura 3 - Dynamic Language Runtime

Esistono alcuni vantaggi dovuti all'utilizzo del DLR, uno dei quali fondamentali per lo scopo di questo studio:

- Il primo serve a semplificare la portabilità dei linguaggi dinamici in .NET Framework. Grazie a DLR e .NET Framework vengono automatizzate molte attività di analisi e generazione del codice. Questo consente ai responsabili dell'implementazione del linguaggio di concentrarsi sulle funzionalità peculiari del linguaggio.
- Il secondo è di abilitare le funzionalità dinamiche nei linguaggi tipizzati staticamente. I linguaggi come C# possono, tramite l'utilizzo di DLR, creare oggetti dinamici e usarli insieme ad oggetti tipizzati in modo statico.
- Il terzo vantaggio, chiave per quanto riguarda IronPython, è di abilitare la condivisione di librerie e oggetti. Gli oggetti e le librerie implementati in un linguaggio possono essere utilizzati da altri linguaggi. DLR consente inoltre l'interoperabilità tra linguaggi tipizzati in modo statico, come C#, e in modo dinamico come Python. Così come C# può dichiarare un oggetto dinamico che usa una libreria scritta in un linguaggio dinamico, anche Python può sfruttare le librerie di .NET Framework.

IronPython: IronPython in Action. Michael J. Foord, Christian Muirhead.
Dynamic Language Runtime: https://docs.microsoft.com/it-it/dotnet/framework/reflection-and-codedom/dynamic-language-runtime-overview

2.2 Compilazione del codice Python

Python, essendo un linguaggio interpretato, non porta alla creazione di un eseguibile come per i linguaggi C o C++. Questo richiede, dunque, al di fuori dell'ambiente di sviluppo, la necessità di avere installato tutto ciò che serve per la corretta esecuzione del codice. La scarsa portabilità del codice dovuta al bisogno di preparare la macchina prima di eseguire programmi Python può portare all'esigenza di un compilatore. L'utilizzo di un programma Python compilato può essere integrato all'interno di altri software. Nel momento in cui un programma è compilato, esso può essere eseguito tramite, ad esempio, un comando bash/cmd. Esistono infatti molti eseguibili che vengono integrati in questo modo all'interno di altri programmi.

2.2.1 Compilazione

Un compilatore è un programma che modifica il codice scritto in un linguaggio di programmazione, detto codice sorgente, in istruzioni appartenenti a un altro linguaggio, detto codice oggetto. Il processo che viene fatto internamente al programma può essere diviso in due passaggi: il *front-end* e il *back-end.* Nella prima fase, il compilatore traduce il linguaggio in un linguaggio intermedio interno al compilatore, vengono fatte quindi diverse tipi di analisi: lessicale, sintattica e semantica. Nella seconda, oltre all'ottimizzazione del codice, avviene il passaggio finale della creazione del codice oggetto.

Esistono diversi tipi di processi di compilazione in base ai servizi che essi possano offrire: si può infatti parlare di *cross compiler* o di *source-to-source compiler*.

Quando si parla di *cross compilazione* si intende un processo secondo il quale il codice sorgente viene compilato in maniera tale da poter essere eseguito su un'architettura diversa da quella in cui il software è stato sviluppato. Lo stesso

principio vale anche per il sistema operativo della macchina sul quale si intende fare eseguire il codice.

Un *source-to-source compiler (S2S)* anche detto *source-to-source translator* è un tipo di compilatore che prende del codice sorgente di un programma scritto in un linguaggio di programmazione e produce un codice oggetto equivalente in un linguaggio di programmazione differente. Il compiler S2S non agisce allo stesso modo di un compilatore convenzionale ma converte codice mantenendo lo stesso livello di astrazione. Un compilatore convenzionale, invece, traduce il codice ad un libello di astrazione più basso.

2.2.2 PyInstaller

PyInstaller è un software in grado di compilare codice scritto in linguaggio Python prendendo tutte le dipendenze del caso e riunendo il tutto in un unico pacchetto con un eseguibile. Successivamente, il codice può essere eseguito su macchine che possono essere sprovviste dell'interprete Python. Un fattore importante è il poter supportare versioni di Python che vanno dalla 3.6 in avanti, inoltre riesce a processare correttamente la maggior parte dei pacchetti come *numpy, PyQt, Django*.

Nonostante il software sia disponibile su tutti i principali sistemi operativi, quali *Windows, Mac Os X* e *GNU/Linux,* esso non va considerato come un cross-compiler. Infatti, l'eseguibile che viene creato è compatibile solamente con il sistema operativo sul quale *PyInstaller* viene eseguito. Questo perché l'interprete Python che viene utilizzato è specifico per quel sistema operativo.

Per quanto riguarda il funzionamento di PyInstaller, esso legge gli script Python che gli sono dati, li analizza e considera ogni libreria di cui essi hanno bisogno per poter eseguire correttamente. Successivamente questi file vengono copiati insieme all'interprete Python e inseriti in una cartella specifica, opzionalmente può essere

creato un singolo file eseguibile. Per poter capire quali siano le librerie che servono per poter eseguire il codice correttamente, PyInstaller legge tutti gli *import* che vengono fatti all'interno dei file e procede ricorsivamente fino ad aver controllato tutta la lista di dipendenze.

Il funzionamento del programma creato ha una particolarità: una volta lanciato l'eseguibile esso non viene eseguito direttamente, ma, al suo posto viene eseguito il *PyInstaller bootloader,* che è un programma binario eseguibile per i sistemi operativi citati in precedenza. Il *bootloader* crea un ambiente Python temporaneo in modo che l'interprete possa cercare tutte le librerie che sono state aggiunte. Una volta completato questo passaggio, la copia dell'interprete esegue normalmente il programma.

2.2.3 Portabilità

Un programma Python compilato tramite il software PyInstaller ha una portabilità limitata al sistema operativo. Tuttavia, il fatto di avere un prodotto che non ha bisogno di nessun tipo di installazione all'interno delle macchine sul quale viene eseguito è un vantaggio non indifferente. Per utilizzare un software Python su una macchina diversa da quella sulla quale si è sviluppato sarebbe necessario installare, oltre all'interprete, tutti i pacchetti necessari. In alternativa è possibile utilizzare un ambiente virtuale come *pipenv* per avere un'automazione di installazione dei pacchetti specifici.

Per poter avere una portabilità garantita su tutte le piattaforme di interesse è necessario ripetere il processo di preparazione delle librerie e di compilazione su tutti i sistemi operativi necessari.

Per quanto riguarda il contesto aziendale nel quale sono collocati i prodotti sviluppati seguendo questo tipo di tecnologia, in alcune situazioni il fatto di avere un eseguibile sembra essere la scelta più adeguata. Un programma in particolare tra quelli sviluppati ha la necessità di poter essere eseguito da operatori che possono non essere qualificati per installare una serie di pacchetti all'interno di una macchina. Tramite l'utilizzo di un singolo file eseguibile l'operazione è più semplice e immediata.

2.3 Differenze

Per chiarire i pregi e i difetti delle due tecnologie trattate in questo capitolo, ovvero il software IronPython e la compilazione, viene fatto un confronto.

L'utilizzo di IronPython è pensato per poter sfruttare le potenzialità di altri linguaggi di programmazione, come un linguaggio di scripting, e di poter sviluppare parti di programma in maniera modulare e interdipendente. Per quanto riguarda l'utilizzo di un compilatore viene data la possibilità di avere una grandissima portabilità e un utilizzo del software, integrato o meno, semplice e senza bisogno di preparazione del sistema.

Un punto importante del confronto è il livello effettivo di integrazione che può essere fatto. Per livello di integrazione si intende quanto i due linguaggi che vengono utilizzati possono essere interconnessi, ovvero a che livello sia possibile utilizzare codice scritto da un linguaggio all'altro, e quindi se possono essere usate classi o se sia possibile unicamente chiamare programmi già completati. Entrambe le tecnologie analizzate verranno quindi valutate anche secondo questo fattore.

2.3.1 Punti di forza di IronPython

Con IronPython si ha un livello di integrazione più alto, ovvero che è possibile utilizzare classi e metodi di classe che sono stati scritti in Python da parte del programma scritto in C#. Questo tipo di integrazione può essere definito come bilaterale, in quanto si possono utilizzare i linguaggi in entrambe le direzioni, Python in C# e viceversa.

L'integrazione risulta complessivamente semplice e l'utilizzo di questo software permette di avere una grande flessibilità in termini di utilizzo modulare di classi o metodi di un linguaggio rispetto all'altro. Utilizzando il software di integrazione rende possibile l'utilizzo delle classi del linguaggio integrato in maniera completa, come se il codice non fosse scritto in un altro linguaggio.

Per poter sviluppare codice Python che deve essere integrato è necessario utilizzare IronPython, il quale gira su una versione *Python3.4*. L'ultima versione di IronPython, uscita in aprile 2021, ha portato a una netta miglioria rispetto alla versione precedente, con un aggiornamento della versione di Python supportata. Le librerie base di Python sono incluse all'interno del software, per altre librerie aggiuntive è necessario controllare la compatibilità.

Gli svantaggi, per quanto riguarda questo tipo di integrazione, sono innanzitutto che per poter sviluppare il codice è necessario utilizzare due IDE differenti: *Visual Studio* per la parte di sviluppo C# e un altro IDE con compilatore IronPython per Python.

Il debug, quando si stanno usando entrambi i linguaggi non è particolarmente utile ed è quindi consigliabile aver testato adeguatamente il codice Python prima di integrarlo all'interno del programma C#.

Nonostante il software includa le librerie più importanti per un utilizzo standard del linguaggio Python, non è garantita la compatibilità con tutte. Questo comporta che

per utilizzi del linguaggio più specifici, come, ad esempio, l'interazione con alcuni database, non sia possibile sviluppare il software nella maniera più indicata.

2.3.2 Punti di forza della Compilazione

La creazione di un eseguibile può essere una buona soluzione per due motivi: il primo è che non sempre la versione di Python su cui si basa IronPython è sufficientemente aggiornata, il secondo, invece, perché esso non supporta alcune librerie.

Per quanto riguarda la prima motivazione, la versione di Python su cui si basa IronPython per il momento è supportata ancora dalla maggior parte delle librerie. Rimane però il fatto che in alcuni casi non sia possibile utilizzare quella specifica versione e si è costretti a passare a versioni più recenti del compilatore come *Python 3.9.x*. Il secondo motivo è il caso in cui una libreria non sia compatibile con IronPython. In alcuni casi, quindi, non sarà possibile installare tale libreria.

Il primo vantaggio dell'utilizzo di una compilazione del codice Python è che non c'è bisogno di utilizzare un software di integrazione per poter utilizzare il codice dall'esterno. Inoltre, non essendo limitati all'utilizzo di una singola versione dell'interprete Python e potendo quindi utilizzare sempre la versione più aggiornata, possiamo sfruttare al meglio tutte le possibilità che il linguaggio ci offre.

Esistono però alcuni contro. Il primo può essere identificato nella difficoltà di mantenimento del codice, in quanto, ogni volta che esso deve essere modificato è necessario creare nuovamente l'eseguibile e integrarlo all'interno del programma principale.

Un altro svantaggio si ha con la perdita della modularità delle classi che appartengono al linguaggio. Utilizzando un eseguibile non è possibile invocare costruttori di determinate classi e utilizzarne i metodi, quindi, il punto di unione fra i due linguaggi si limita ad essere il passaggio di parametri tramite la chiamata dell'eseguibile da parte di una *command line*.

Parte II

Software integrato in contesto aziendale

Capitolo 3

Software Sviluppato

3.1 Software per estrazione di Informazioni

Un servizio dato ai clienti dell'azienda è un'analisi di fattibilità per quanto riguarda l'aggiornamento di alcuni software all'interno delle stampanti. Ogni richiesta di assistenza prevedeva quindi la necessità di un collegamento in remoto sulle macchine di interesse. Una volta collegati vi era un passaggio di installazione ed esecuzione di un software per l'estrazione di informazioni. Altre informazioni necessarie, come la versione dei driver delle schede di rete, non venivano individuate dal software utilizzato e dovevano essere ricercate manualmente.

Nasce così la necessità di sviluppare un software in grado di effettuare queste operazioni in autonomia e di poter serializzare i risultati in diversi formati.

Tramite l'utilizzo di questo software viene meno la necessità di dover effettuare questo tipo di assistenza. Il cliente, al posto di dover potenzialmente fermare la produzione della stampante per il tempo necessario all'assistenza, può far raccogliere le informazioni da qualsiasi operatore in autonomia e allegarne i risultati alla richiesta di analisi di fattibilità.

3.1.1 Analisi e progetto del software

Secondo le necessità elencate sopra, il software deve occuparsi di estrarre le principali informazioni riguardo al computer sul quale viene eseguito. Le informazioni in questione descrivono le componenti software della macchina:

- Sistema operativo:
    - Tipo di sistema operativo
    - Versione
    - Codice di attivazione del sistema
    - Architettura
    - Numero di build
- Schede di rete:
    - Nome della scheda
    - Informazioni sull'indirizzo IP
    - Indirizzo MAC
- CPU:
    - Nome
    - Tipo di CPU
- GPU:
    - Nome
- Scheda madre:
    - Produttore
    - Tipo di scheda madre
    - Versione
    - Numero seriale
- Driver delle schede di rete:
    - Nome del produttore
    - Nome del driver
    - Versione

Queste informazioni vengono serializzate in due formati: TXT e XML. Il primo formato è stato pensato per avere un'immediata comprensione dei dati, il secondo, invece, per poter essere utilizzato da altri software. Un fattore importante era di avere un sistema che potesse funzionare su stampanti collocate in diversi stati. Non potendosi affidare ad un linguaggio singolo per tutte le macchine, i comandi lanciati dal prompt utilizzato dal software non potevano avere opzioni scritte in italiano.

3.1.2 Operazioni principali

Le operazioni effettuate dal software seguono il flusso descritto in figura 4.

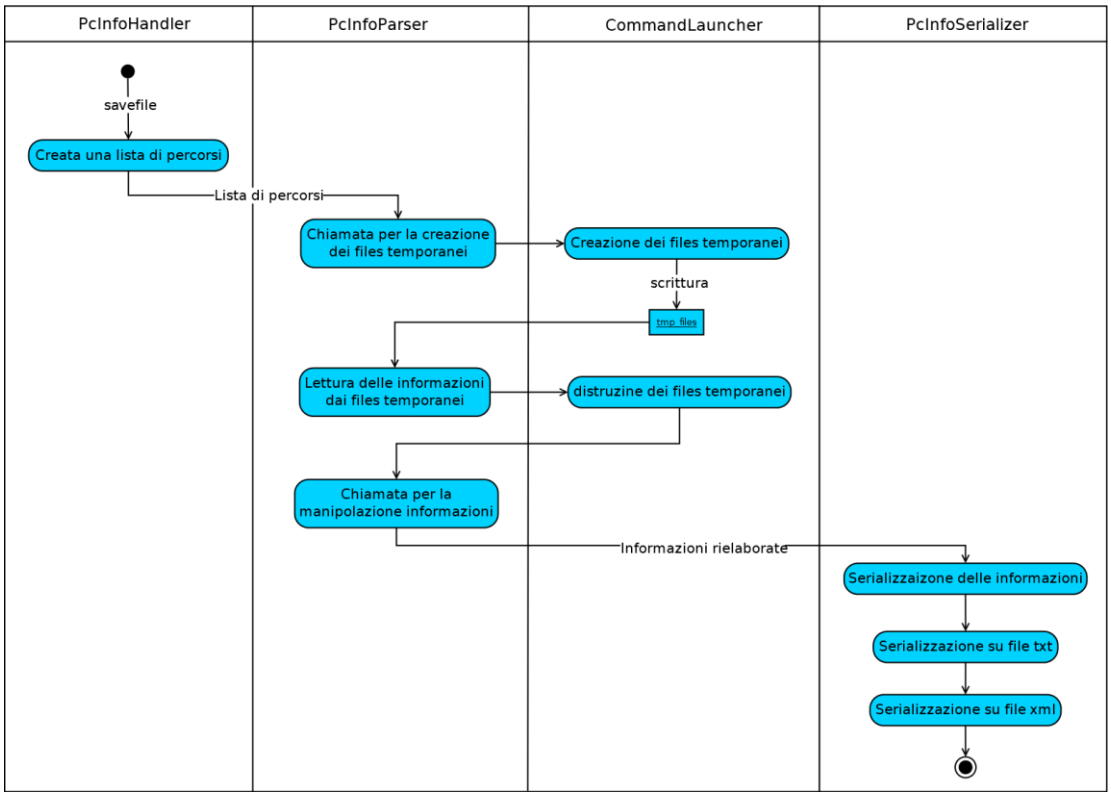

Figura 4 – Flow diagram Estrattore

All'interno del diagramma si possono notare le classi principali del software. La prima, in ordine di esecuzione, è *PcInfoHandler*, nel quale viene creata una lista di percorsi che serviranno come riferimento per poter salvare temporaneamente le informazioni estratte. La lista di percorsi viene presa dalla classe *PcInfoParser* e mandata a *CommandLauncher*. Questa ultima classe si occupa di effettuare una serie di comandi cmd per poter estrapolare varie informazioni dalla macchina. Le informazioni sono salvate all'interno di alcuni file temporanei, i quali dovranno subire un processo di *parsing*, mantenendo solamente i contenuti importanti. Terminato il processo di *parsing* i file temporanei vengono eliminati dalla macchina e le informazioni specifiche vengono mandate sotto forma di dizionari alla classe *PcInfoSerializer*. La classe in questione serializza le informazioni all'interno di un

file TXT, mentre utilizza una libreria apposita per poter scrivere informazioni strutturate in XML.

3.1.3 Librerie utilizzate

Le librerie utilizzate per questo progetto erano tutte compatibili con l'interprete IronPython ed è stato quindi possibile utilizzarlo per sviluppare il programma. Come si può notare dal diagramma numero 5, le librerie meno comuni che sono state utilizzate sono state *xml* e *pathlib*.

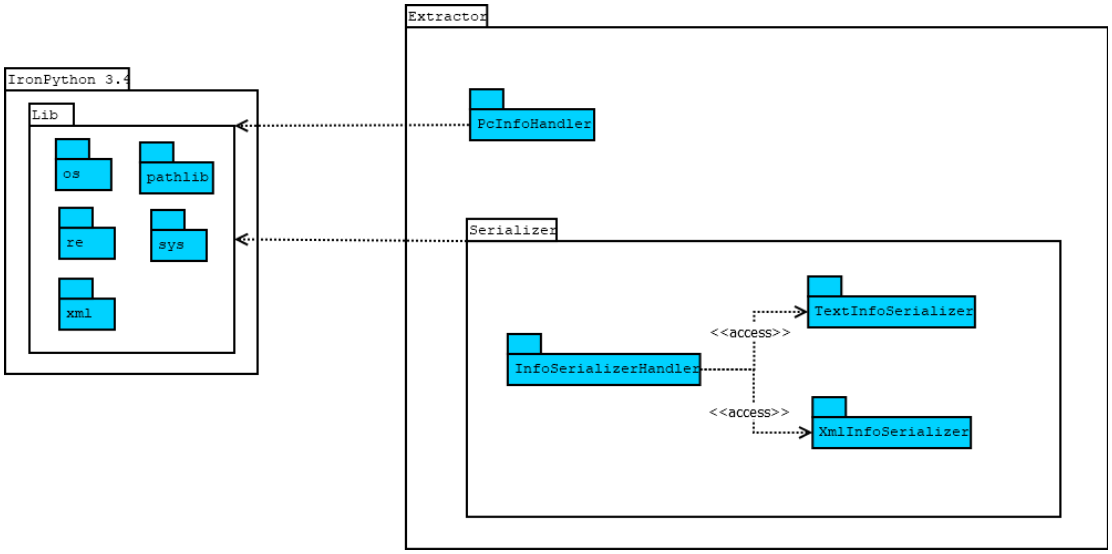

Figura 5 – Package diagram Estrattore

xml è una libreria Python usata per creare file in formato XML. Questo formato viene utilizzato per avere un tipo di informazione che sia strutturata e utilizzabile tramite apposite API anche da altri software.

La seconda libreria utilizzata è chiamata *pathlib*, usata per poter utilizzare in maniera più agile i percorsi del filesystem, con la possibilità di adattare i percorsi a filesystem

Unix o Windows. Un metodo interessante di questa libreria è detto *suffix*, usato per estrarre l'estensione di un file senza dover ricorrere all'utilizzo delle espressioni regolari.

3.1.4 Risultati ottenuti

Il software, tramite una serie di comandi lanciati da un prompt, riesce a raccogliere tutte le informazioni e a serializzarle nei due tipi di file. Le macchine sulle quali è stato utilizzato questo software hanno come sistema operativo Windows. Per verificare il corretto funzionamento del software, esso è stato testato su delle stampanti in produzione e i risultati sono stati soddisfacenti. Il software non ancora integrato ha prodotto dei risultati corretti nonostante la macchina avesse caratteristiche hardware molto diverse da quella sul quale il programma è stato sviluppato.

```
<DRIVERS>
  <deviceclass>NET</deviceclass>
  <driverversion>12.18.11.1</driverversion>
  <driverprovidername>Intel</driverprovidername>
  <devicename>Intel(R) I210 Gigabit Network Connection</devicename>
  <location>PCI bus 13, device 0, function 0</location>
</DRIVERS>
<DRIVERS2>
  <deviceclass>NET</deviceclass>
  <driverversion>12.18.11.1</driverversion>
  <driverprovidername>Intel</driverprovidername>
  <devicename>Intel(R) I210 Gigabit Network Connection</devicename>
  <location>PCI bus 12, device 0, function 0</location>
</DRIVERS2>
```

Figura 6 – Output XML Estrattore

Il formato XML in figura 6 mostra in dettaglio solo alcune caratteristiche relative ai driver, nel formato TXT in figura 7 si può notare che le stesse informazioni riportate sono di più facile comprensione.

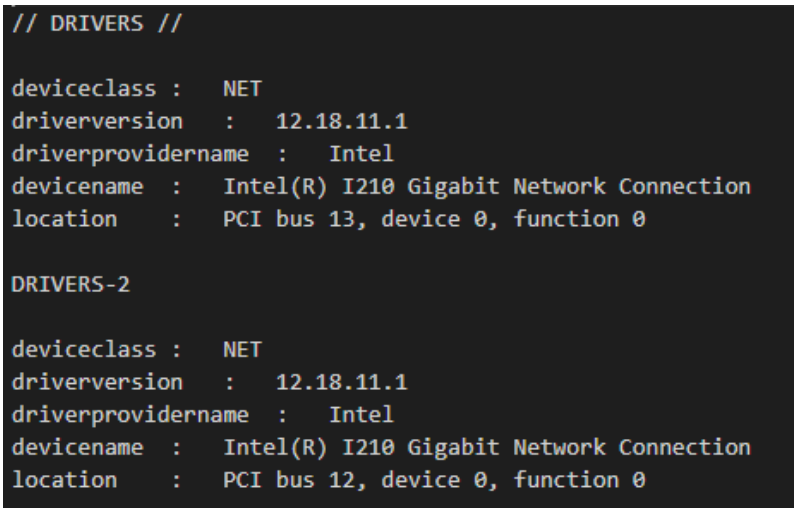

Figura 7 – Output TXT Estrattore

3.1.5 Operazioni di integrazione

L'integrazione di questo programma all'interno di un altro è stata possibile con entrambi i meccanismi analizzati nel capitolo 2.

Il primo metodo di Integrazione è stato tramite l'utilizzo di IronPython, il secondo tramite il compilatore PyInstaller. Per quanto riguarda le prestazioni dei due metodi, essi verranno analizzati all'interno del capitolo 4. Il fatto di aver utilizzato esclusivamente librerie compatibili ha reso possibile l'integrazione tramite IronPython.

L'integrazione è stata fatta per poter inserire il software Python all'interno di un programma più grande scritto interamente in C#. Questo programma è stato creato dall'azienda per poter automatizzare alcuni processi di assistenza e setup delle stampanti.

Il processo di integrazione tramite IronPython è composto da tre passaggi fondamentali, evidenziati in figura 8.

```
class PythonScriptLink
{
    public static string _currentPath = Directory.GetCurrentDirectory();
    0 references
    static void Main(string[] args)
    {
        string currentDir = _currentPath + @"\python_scripts\";
        string sourceScript = currentDir + "PcInfoHandler.py";
        string destPath = currentDir + "fileprova.txt";
        string libDir = @"C:\Program Files\IronPython 3.4\Lib";

        List<string> variablesList = new List<string>();
        variablesList.Add(currentDir);
        variablesList.Add(libDir);                                  1

        try
        {
            var engine = Python.CreateEngine();

            var source = engine.CreateScriptSourceFromFile(sourceScript);
            var scope = engine.CreateScope();
            engine.SetSearchPaths(variablesList);

            source.Execute(scope);                                  2

            // Variable injection

            dynamic Class = scope.GetVariable("PcInfoHandler");
            dynamic Class_instance = Class(destPath);
            Class_instance.pc_info_handler();                       3

        }
        catch(Exception ex)
        {
            Console.WriteLine("Error : " + ex.Message);
        }


        Console.Read();

    }
}
```

Figura 8 – Integrazione Estrattore

Passaggio 1 - Preparazione dei percorsi

I percorsi principali che servono per poter eseguire gli script e sono tre:

- Directory nella quale sono posizionati gli script Python.
- Classe Python della quale si vuole effettuare l'integrazione.
- Cartella con le librerie di IronPython, necessarie per le dipendenze dello script Python che verrà eseguito.

Passaggio 2 - Creazione dell'engine

Una volta impostati correttamente tutti i riferimenti agli script di Python che abbiamo intenzione di utilizzare, possiamo estrarre, tramite l'*engine,* quello che ci serve. Viene creato quindi un singolo *engine*. La *DLR* supporta la creazione di più contesti di esecuzione all'interno di una singola applicazione. Questo si rivela utile per la creazione di contesti di esecuzione che siano isolati gli uni dagli altri. Una volta creato l'*engine* è possibile eseguire il codice.

Per eseguire il codice si devono introdurre due componenti, detti *ScriptSource* e *ScriptScope*. "Il primo viene creato dall'engine a partire da dei file, come nel caso in esame, oppure da uno script passato come stringa. Lo *ScriptScope* rappresenta un *namespace*, anch'esso creato dall'engine. [4]"

Viene infine eseguito lo scope creato tramite l'engine.

Passaggio 3 - Variable injection e utilizzo classi Python

A questo punto è possibile utilizzare le classi Python mediante l'utilizzo di un meccanismo detto *Variable injection*. Tramite i metodi *GetVariable* e *SetVariable* è possibile dialogare liberamente con le classi Python come se fossero classi C#. Con il metodo *GetVariable* dello scope creato in precedenza si può fare riferimento al nome della classe che si vuole utilizzare. A questo punto è possibile utilizzare i metodi legati alla classe.

3.2 Software per interazione con database

Lo sviluppo di questo software è stato pensato per poter automatizzare il processo di pulizia di un database detto *shrinking.* Il processo è richiesto dai clienti nel caso in cui le stampanti subiscano seri rallentamenti al software di gestione della stampa. I rallentamenti sono causati da un problema legato a meccanismi interni alla macchina che causano una scrittura eccessiva all'interno di una colonna di un database.

Lo *shrinking* di un database di questo tipo consiste nell'eliminazione di una colonna specifica che, se sovraffollata, causa il rallentamento dell'intero sistema. Nel momento in cui la colonna viene alleggerita, il sistema dei database Firebird ha bisogno di un passaggio di *backup* e *restore* per rendere effettivo l'alleggerimento. Quindi, prima di poter vedere la grandezza del database diminuire di un valore significativo, non basta eliminare le informazioni all'interno della colonna specifica.

3.2.1 Analisi e Progetto del software

Il problema che causa l'eccessiva grandezza del database è causato da meccanismi interni alla stampante che provocano una sovrascrittura dei record di log ogni pochi secondi. Questo provoca la scrittura di un'enorme mole Log all'interno della tabella *TAB_Logs*.

Il software progettato deve effettuare tutte le operazioni di *shrinking* elencate in precedenza in piena autonomia. Esso parte dal controllo sull'eccessiva grandezza del database, viene fatta la pulizia dei log in un intervallo di tempo maggiori di un numero di giorni dato dal cliente e infine viene fatto il backup e il restore del database.

È necessario specificare che il database, in tutte le stampanti che vengono date ai clienti, si trova nello stesso percorso e ha lo stesso nome; quindi, è possibile utilizzare

un percorso statico all'interno dell'intero programma. Un'altra precisazione riguarda il modus operandi del programma. Utilizzando C# non ci sono librerie o API che permettono di fare direttamente il meccanismo di backup e di restore, questo procedimento rendeva necessario l'utilizzo di comandi tramite un prompt. Con l'utilizzo di una libreria Python, invece, questa operazione può essere fatta tramite l'utilizzo di una libreria apposita.

Lo scopo della creazione di questo software, come per il precedente, è di agevolare il processo di assistenza, dando autonomia all'operatore di poter migliorare le prestazioni della macchina senza dover fermare la macchina per effettuare il processo di assistenza.

3.2.2 Operazioni principali

Osservando il diagramma di flusso in figura 9 possiamo notare le classi principali che compongono il software.

La prima classe che viene eseguita, *DbShrinkerHandler*, si occupa di effettuare la connessione con il database, avendo salvate le credenziali standard per l'accesso a tutti i database. La connessione viene garantita dall'API chiamata *fdb*, tramite la quale verranno fatte tutte le operazioni fondamentali. Una volta garantita la connessione con il database, un'istanza della classe, chiamata *Launcher*, viene passata alla classe *DbAnalyzer*, la quale, cerca, utilizzando la classe *QueryLauncher*, i nomi di tutte le tabelle del database e ne calcola la grandezza. Se la grandezza della tabella di interesse è maggiore di una soglia data, allora inizia il processo di pulizia.

La gestione della pulizia vera e propria viene fatta dalla classe *Shrinker*. Questa classe manda, tramite *fdb*, la query per poter eliminare i log più vecchi di una certa data, con default impostato a 30 giorni. Una volta eliminati i log viene creato un

nuovo database come backup dell'originale. Durante questa operazione si opera anche alla restore del database, tramite la quale l'alleggerimento risulta definitivo.

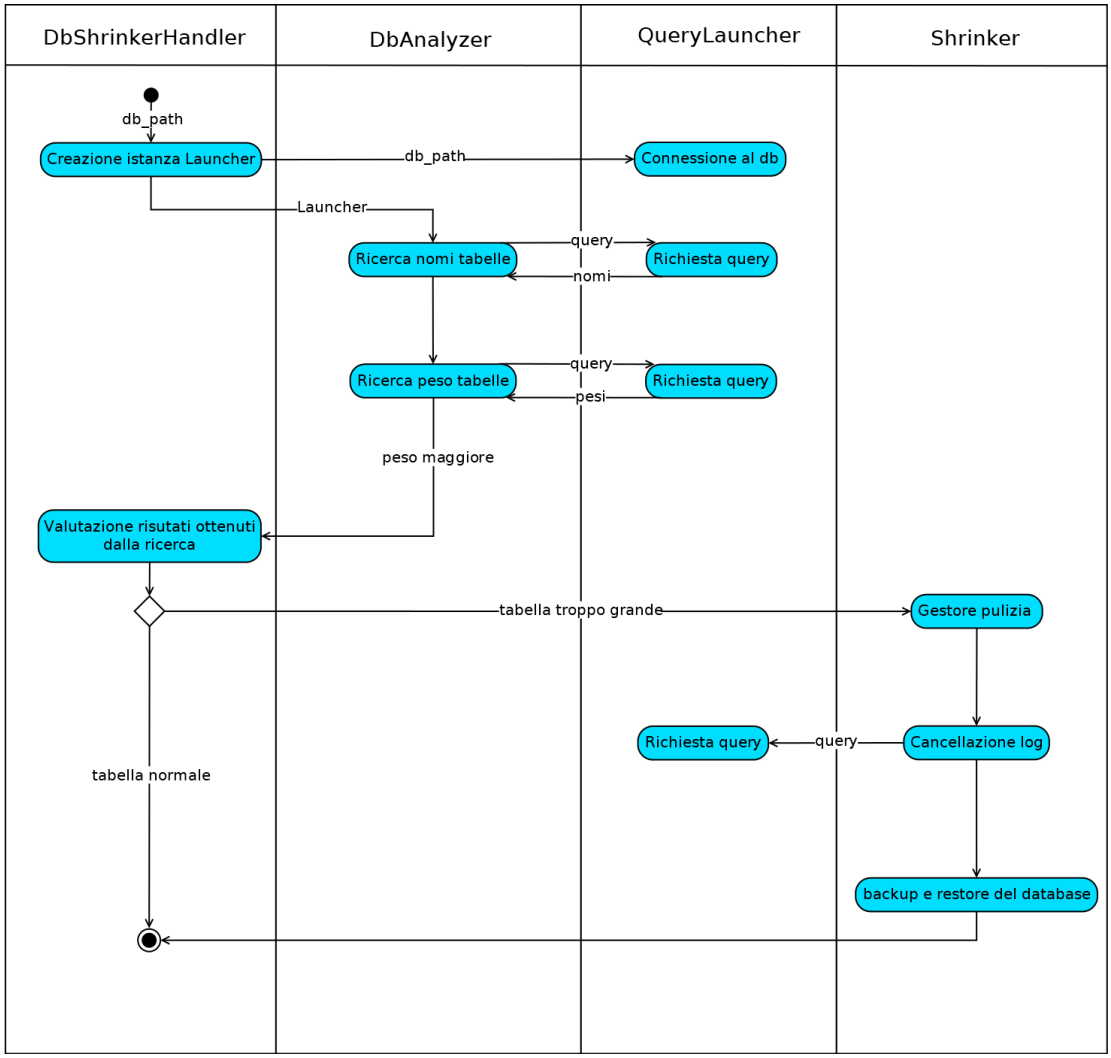

Figura 9 – Flow diagram Shrinker

3.2.3 Librerie utilizzate

Le librerie utilizzate sono state *re* e *fdb*.

La prima libreria, *re*, viene utilizzata per gestire le espressioni regolari. Fornisce quindi metodi per poter interagire con stringhe e poter selezionare stringhe che fanno match con delle espressioni regolari.

La seconda libreria, *fdb*, a differenza di *re* non è largamente utilizzata, in quanto si occupa dell'interazione con i database Firebird. Questa libreria, infatti, è implementata per garantire un supporto alla connessione con i database relazionali Firebird. Oltre all'insieme di funzionalità standard che danno altre API Python per i database, *fdb* espone delle API per ulteriori operazioni. Quindi, oltre alla possibilità di poter effettuare delle query sul database, c'è la possibilità di effettuare altre operazioni, come quelle elencate precedentemente.

Il problema maggiore, incontrato utilizzando questa libreria, è stato l'impossibilità di utilizzare IronPython, in quanto *fdb* è incompatibile con il suo engine. Nonostante siano state testate altre librerie, quali *firebirdsql* e *sqlalchemy*, non è stato possibile utilizzare il software di integrazione nello sviluppo di questo programma. Si è rivelato di fondamentale importanza utilizzare il secondo sistema di integrazione.

3.2.4 Risultati ottenuti

I risultati ottenuti sono stati soddisfacenti e c'è stato quindi un incremento della velocità per quanto riguarda questo tipo di assistenze. Il programma, una volta fatto girare su una macchina, riesce velocemente, in proporzione alla grandezza del database, ad effettuare il meccanismo di pulizia, prendendo il database e sostituendolo con un suo backup più leggero.

La creazione di questo software, oltre che a velocizzare il meccanismo di pulizia, può portare a degli sviluppi futuri, quali l'utilizzo del software in maniera periodica per evitare in futuro la comparsa di questo tipo di problema. Un altro impiego potrebbe essere sempre l'utilizzo del software ma tramite un meccanismo a soglia, superata la quale entra in funzione la pulizia del database. È importante notare che il risparmio di tempo durante questo tipo di interventi, o addirittura la completa eliminazione di interventi di questo tipo, porta ad un risparmio per i clienti, in quanto alcune operazioni che vengono fatte sulle stampanti comporta un arresto della produzione per il tempo necessario alla risoluzione del problema.

3.2.5 Operazioni di integrazione

L'integrazione, come precedentemente accennato, è stata fatta seguendo il metodo della compilazione. Il motivo principale è stata l'incompatibilità della libreria *fdb* con IronPython, senza la quale non sarebbe stato possibile creare il software di pulizia. Seguendo quindi questo metodo, è stato necessario utilizzare PyInstaller, tramite il quale creare un singolo eseguibile da poter chiamare dal programma principale tramite l'utilizzo di un prompt. Il software è stato sviluppato in maniera tale da poter supportare l'utilizzo di parametri per la chiamata, in assenza dei quali utilizzare parametri di default.

Il primo passaggio per poter effettuare questo tipo di integrazione è la creazione dell'eseguibile tramite PyInstaller. La sintassi è la seguente:

```
pyinstaller [-n <nome_eseguibile>] <script da rendere eseguibili> [--noconsole] [-- onefile]
```

- L'opzione *-n <nome_eseguibile>* serve per decidere il nome da dare all'eseguibile, se non viene specificata questa opzione, l'eseguibile che viene

creato sarà chiamato con il nome del primo file che viene inserito nella lista di script.

- L'opzione *–onefile* serve per creare l'eseguibile in un file singolo.

Il secondo passaggio, come è possibile notare in figura 10, è la chiamata dell'eseguibile a partire dal programma in C#.

```
string _currentPath = Directory.GetCurrentDirectory();
string currentDir = _currentPath + @"\python_scripts\";
                                                        1

string command = $" {currentDir}Shrin.exe";
Process process = new Process();
ProcessStartInfo startInfo = new ProcessStartInfo();
startInfo.WindowStyle = ProcessWindowStyle.Hidden;
startInfo.FileName = command;
startInfo.Arguments = destPath;
process.StartInfo = startInfo;
process.Start();
process.WaitForExit();                                  2
```

Figura 10 – Integrazione Shrinker

La prima operazione è la preparazione dei percorsi per poter utilizzare il software.

Per poter utilizzare lo script Python è necessario creare un nuovo processo, ne viene creato uno e ad esso viene dato il comando da eseguire assieme agli argomenti necessari al corretto funzionamento del programma. Il metodo *WaitForExit* serve per attendere che il programma abbia finito di eseguire ed è quindi opzionale. Questo metodo è stato utilizzato per poter effettuare correttamente le misurazioni per verificare anche le prestazioni del software.

Capitolo 4

Misurazioni e risultati ottenuti

4.1 Meccanismo di misurazione delle performance

Per poter fare un confronto tra i due metodi di integrazione, oltre all'aver osservato i limiti di entrambi, essi possono essere valutati anche a livello prestazionale. Per livello prestazionale si intende la velocità di esecuzione.

Per poter testare la velocità di esecuzione dei due meccanismi è stato utilizzato l'unico software con il quale è stato possibile effettuare entrambi i metodi di integrazione. Il software in questione si occupa della raccolta e serializzazione di informazioni riguardanti il computer e, non utilizzando nessuna libreria specifica, è stato possibile effettuare i test prestazionali. Per misurare la velocità del software sviluppato prima dell'integrazione è stato utilizzato un metodo implementato in linguaggio Python, mentre per vedere l'effettiva differenza tra i due metodi di integrazione è stata utilizzata una classe C# detta *StopWatch.*

Le misurazioni sono state ripetute cinque volte per evitare che alcuni processi interni alla macchina potessero contaminare i risultati. In tal senso sono state prese delle precauzioni cercando di arrestare altri processi interni alla macchina e chiudere ogni connessione con l'esterno. Una volta ottenute le misurazioni è stata fatta una media dei risultati.

Prima di poter parlare di misurazioni vere e proprie è necessario mostrare le specifiche software e hardware della macchina sulla quale il programma è stato fatto eseguire. All'interno della tabella in figura numero 11, è possibile vedere le principali informazioni riguardo alla macchina.

Caratteristiche	
Sistema operativo	Windows 10 Pro
Architettura	64 bit
Cpu	Intel Core i7-5820K
Clock	3.3 GHz
Numero di core	6
Gpu	NVIDIA GeForce GT 710
Ram	8 Gb
Memoria	SSD da 500 GB

Figura 11 – Tabella specifiche hardware software

4.1.1 Misurazione nel codice Python

La misurazione delle prestazioni del primo metodo è stata fatta tramite l'utilizzo di un decoratore Python.

Un *decoratore* è una funzione che prende come parametro un'altra funzione, aggiunge delle funzionalità e ne restituisce un'altra, senza alterare il codice sorgente della funzione passata come parametro.

È possibile utilizzare questo meccanismo in quanto per Python le funzioni sono *First Class Objects*, ovvero che possono essere passate come parametro, assegnate a variabili e restituite come qualsiasi altro valore. Possono essere inoltre definite all'interno di altre funzioni, nel qual caso si parla di funzioni annidate.

In quanto non era necessario testare una singola funzione ma l'esecuzione dell'intero programma, la funzione decorata è stato il *main* del programma.

```python
def timing(function):
    """Funzione per misurare il tempo di esecuzione di un processo

    Questa funzione annidata è utilizzabile come decoratore:

    @timing
    def do_work() :
        codice
    """
    def wrap(*args, **kwargs):
        time1 = time.time()
        ret = function(*args, **kwargs)
        time2 = time.time()
        gap = (time2 - time1)*1000.0
        print(f'\nLa funzione {function.__name__} ci ha messo {gap} ms\n')
        return ret
    return wrap
```

Figura 12 – Decoratore timing Python

Tramite quindi l'utilizzo dei decoratori è possibile aggiungere funzionalità a funzioni già complete senza alterarne il codice interno. Come è possibile notare in figura 12, il decoratore utilizzato è stato chiamato timing ed è stato applicato al codice del software. Questo decoratore prende in ingresso la funzione da cronometrare e salva l'orario nel momento in cui inizia l'esecuzione. Quando il programma ha terminato l'esecuzione salva il tempo in quell'istante e ne stampa la differenza dopo averla calcolata.

4.1.2 Misurazione nel codice C#

Per poter misurare le prestazioni dei software in C# si è utilizzato un meccanismo simile a quello utilizzato per Python.

La classe utilizzata per effettuare le misurazioni è stata *StopWatch*. Come è possibile notare in figura 13, dopo aver creato un'istanza della classe *StopWatch* è stato

chiamato il metodo *Start()* appena prima dell'esecuzione del programma per poter iniziare la misurazione cronometrica. Una volta terminata l'esecuzione, il cronometro è stato fermato tramite l'utilizzo del metodo *Stop()* e il risultato è stato stampato in millisecondi tramite *ElapsedMilliseconds.*

```
Stopwatch stopwatch1 = new Stopwatch();
stopwatch1.Start();
IronPythonScript ironScript = new IronPythonScript();
ironScript.IronStartScript();
stopwatch1.Stop();
Console.WriteLine("Tempo di esecuzione del programma integrato: {0} ms",
    stopwatch1.ElapsedMilliseconds);
```

Figura 13 – C# Stopwatch

L'utilizzo delle misurazioni cronografiche effettuate tramite C# è stato fondamentale per poter arrivare a delle conclusioni su quale metodo di integrazione fosse il più prestazionale.

4.2 Misurazioni ottenute

Le misurazioni che sono state fatte hanno coinvolto entrambi i metodi di integrazione e sono state fatte utilizzando la classe *StopWatch* vista precedentemente.

I test, per evitare un'alterazione dei risultati dovuta a processi interni alla macchina, sono stati ripetuti cinque volte. Sempre per lo stesso motivo si è cercato di chiudere la maggior parte dei processi e di isolare la macchina da connessioni esterne.

Una volta ottenute tutte le misurazioni, le quali, come previsto, avevano delle oscillazioni dovute ai meccanismi interni alla macchina, ne è stata fatta una media. I

risultati hanno definito in maniera chiara quale dei due meccanismi fosse il più prestazionale.

Prima di poter arrivare a un confronto vero e proprio dei risultati è necessario fare una premessa. Nella parte uno si è parlato delle differenze tra i due metodi di integrazione e dei meccanismi interni ad essi. IronPython sfrutta il meccanismo della *DLR*, ovvero un meccanismo di interpretazione, mentre PyInstaller esegue una vera e propria compilazione del codice.

In questo capitolo, dove le premesse teoriche sono state applicate alla pratica, si dovrebbe trovare conferma di quanto detto all'interno dei primi due capitoli.

I risultati ottenuti hanno confermato tutti i presupposti precedenti. Nel grafico di confronto in figura 14, si può notare come IronPython, avendo bisogno dell'interpretazione del codice, sia meno prestazionale dell'utilizzo di un eseguibile.

Le misurazioni sono state fatte in millisecondi

IronPython	Eseguibile
8013,00	4337,00
7874,00	3803,00
8415,00	3809,00
8270,00	3675,00
8082,00	4978,00
Media :	Media :
8130,8	**4120,40**

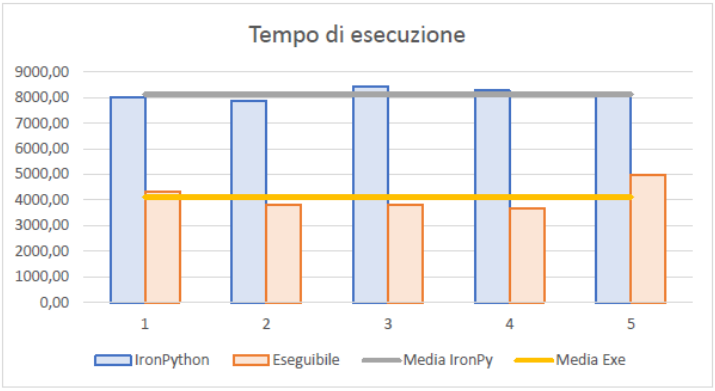

Figura 14 – Confronto delle misurazioni ottenute

La linea gialla rappresenta la media delle misurazioni effettuate utilizzando l'eseguibile mentre quella grigia è la media delle misurazioni effettuate tramite l'utilizzo di IronPython. Questo tipo di studio ha confermato non solo che l'eseguibile è risultato essere più veloce, ma lo è stato di un 50% in più.

Capitolo 5

Conclusioni

5.1 Considerazioni finali

Attraverso questo studio si è potuto chiarire in che modo e con quali mezzi è possibile integrare due software scritti in linguaggi di programmazione diversi. Dopo aver analizzato il processo in ambito teorico si è arrivati ad un approccio pratico all'argomento, scoprendo alcune divergenze tra la teoria e la pratica.

La principale differenza tra la teoria presentata precedentemente e la realtà dei fatti è stata la scoperta dell'incompatibilità di alcune librerie Python con il software di integrazione IronPython. Questo fatto ha portato alla ricerca di un secondo metodo di integrazione, il quale si è rivelato più prestazionale.

Arrivati a questo punto sorge spontanea la domanda se sia veramente utile utilizzare un meccanismo di integrazione all'interno dello sviluppo software e quale dei due meccanismi sia il più efficace.

5.2 Il meccanismo di integrazione è utile?

Il meccanismo di integrazione può essere utile per alcuni motivi. In questo studio non è stato affrontato tale argomento, ma, oltre alla possibilità di aggiungere codice Python a linguaggio C#, è possibile effettuare l'operazione inversa. Un esempio che può rendere più chiaro l'enorme potenziale dell'integrazione è l'utilizzo semplice, cosa garantita dalla semplicità del linguaggio Python in sé, delle API per poter utilizzare il software Microsoft Excel. Tramite l'utilizzo di pochissime righe di codice è possibile creare tabelle, popolarle ed effettuare operazioni su di esse.

Tornando allo studio fatto, ovvero l'integrazione effettuata in senso opposto, si può affermare che questa tecnologia possa essere di grande aiuto in quanto riesce a conciliare le potenzialità di entrambi i linguaggi.

In conclusione, conviene utilizzare il software IronPython? Sì, ma a determinate condizioni. Prima di poter utilizzare IronPython è necessario valutare se tutte le librerie che si ha intenzione di utilizzare siano compatibili con il software, cosa che potrebbe addirittura rendere necessario l'utilizzo di versioni *legacy* di tali librerie. Oltre a questo motivo non ci sono ragioni per le quali IronPython non debba essere utilizzato.

Per quanto riguarda l'integrazione, indipendentemente dal software IronPython, essa è stata fondamentale per l'automazione del processo di *shrinking*, in quanto, senza poter fare affidamento sulla libreria *fdb*, il processo di pulizia dei database sarebbe stato meno semplice e sicuramente meno prestazionale.

5.3 Qual è il meccanismo più efficiente?

Non è possibile considerare un singolo approccio all'integrazione come migliore dell'altro, in quanto, come già affrontato nel capitolo 2, entrambi i metodi hanno i loro pregi e i loro difetti. Possiamo però arrivare a definire in modo più concreto come i due metodi possano essere differenti.

Tramite le misurazioni delle prestazioni è sato confermato come il metodo della compilazione sia stato il più prestazionale, ma questo non è l'unico parametro da considerare. Nonostante le prestazioni siano importanti, un secondo aspetto da valutare è quanto sia effettiva l'integrazione.

Considerando il tempo di esecuzione dei due approcci, è conveniente effettuare la creazione di un eseguibile. Questo, però, nel caso in cui le operazioni che devono

essere fatte da tale software siano statiche, ovvero nel momento in cui non ci siano informazioni che devono essere passate in modo bidirezionale tra i due software. Infatti, il software Python che è stato testato all'interno del capitolo tre aveva il compito di effettuare una mera serializzazione di informazioni su file, ovvero un'operazione che non necessita di nessuna particolare interazione con il software chiamante. Nel momento in cui viene utilizzato un eseguibile, non viene data un'informazione di ritorno al software chiamante, cosa che pone un vincolo importante rispetto a quella che è l'interazione tra i due software.

L'utilizzo di IronPython è, come già detto, per definizione meno prestazionale rispetto all'utilizzo di un eseguibile, ma rende possibile una vera integrazione fra i due software.

In definitiva si può affermare che, per operazioni che non necessitano dell'utilizzo di parametri che devono essere ritornati al software chiamante, la scelta migliore è quella di utilizzare il meccanismo della compilazione. Se, invece, c'è bisogno di un'integrazione completa, è più conveniente utilizzare IronPython perché esso è l'unico metodo che garantisca una vera integrazione.

Bibliografia

[1] Wikipedia : https://it.wikipedia.org/wiki/Python

[2] Python.it : https://www.python.it/about/

[3] Wikipedia : https://it.wikipedia.org/wiki/C_sharp

[4] IronPython in Action. Michael J. Foord, Christian Muirhead.